

Harold Abelson  
Gerald Jay Sussman  
Julie Sussman

---

Struktura  
i interpretacja  
programów  
komputerowych

Słowo wstępne Alana J. Perlisa

---

Z angielskiego przełożył  
Marcin Kubica

---

## O Autorach:

**Harold Abelson** ma stopnie bakałarza (Princeton University) i doktora matematyki (MIT).

Jest profesorem na Wydziale Elektrotechniki i Informatyki w Massachusetts Institute of Technology. Od wielu lat zajmuje się wykorzystaniem komputerów w dydaktyce. Kierował pracami nad pierwszą implementacją języka Logo na Apple II. W 1981 roku napisał z Andreą diSessą książkę „Turtle Geometry” (wydaną po polsku przez Wydawnictwa Naukowo-Techniczne pt. „Geometria żółwia”), przedstawiającą zastosowanie komputera do nauki geometrii, przyczyniając się tym samym do rewolucyjnych zmian w całym procesie nauczania/uczenia się.

Za swoje osiągnięcia otrzymał kilka nagród, między innymi Amar G. Bose Award, przyznaną przez MIT (1992 r.), oraz Taylor L. Booth Education Award, przyznaną przez IEEE Computer Society (1995 r.).

Jest członkiem stowarzyszenia IEEE.

**Gerald Jay Sussman** ma stopnie bakałarza i doktora matematyki (MIT, 1968 r. i 1973 r.).

Jest profesorem na Wydziale Elektrotechniki i Informatyki w Massachusetts Institute of Technology. Od 1964 roku brał udział w badaniach nad sztuczną inteligencją w MIT. Zajmował się też językami programowania, architekturą komputerów i projektowaniem układów VLSI. W 1975 roku razem z Guyem L. Steele'em, Jr. opracował język programowania Scheme. Był też głównym konstruktorem maszyny Digital Orrery, wykorzystywanej do badania ruchu planet w Układzie Słonecznym, i wieloprocesorowego komputera Supercomputer Toolkit, zoptymalizowanego pod kątem opracowywania układów równań różniczkowych zwyczajnych.

Za swoje osiągnięcia na polu nauczania informatyki otrzymał nagrody: Karl Karlstrom Outstanding Educator Award, przyznaną przez ACM (1990 r.), i Amar G. Bose Award, przyznaną przez MIT (1991 r.).

Jest członkiem stowarzyszeń: IEEE, NAE, AAAI, ACM. Należy też do American Academy of Arts and Sciences. Z zamiłowania jest słusarzem. Jest członkiem organizacji zrzeszających zegarmistrzów.

**Julie Sussman** pisze i redaguje teksty w języku angielskim, a także w językach programowania.

Dane o oryginalu

HAROLD ABELSON and GERALD JAY SUSSMAN with JULIE SUSSMAN

## **Structure and Interpretation of Computer Programs**

Second Edition

© 1996 by The Massachusetts Institute of Technology

Second Edition

Prowadzenie serii *Elżbieta Beuermann*

Redaktorzy *Izabela Ewa Mika, Ewa Zdanowicz*

Okładkę i strony tytułowe projektował *Paweł G. Rubaszewski*

Redaktor techniczny *Barbara Chojnacka-Flisiuk*

Korekta *Zespół*

Skład i łamanie *preTeXt*

Podręcznik akademicki dotowany przez Ministerstwo Edukacji Narodowej i Sportu

© Copyright for the Polish edition

by Wydawnictwa Naukowo-Techniczne

Warszawa 2002

All Rights Reserved

Printed in Poland

Utwór w całości ani we fragmentach nie może być powielany ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych, kopiujących, nagrywających i innych, w tym również nie może być umieszczany ani rozpowszechniany w postaci cyfrowej zarówno w Internecie, jak i w sieciach lokalnych bez pisemnej zgody posiadacza praw autorskich.

Adres poczty elektronicznej: [wnt@pol.pl](mailto:wnt@pol.pl)

Strona WWW: [www.wnt.com.pl](http://www.wnt.com.pl)

ISBN 83-204-2712-6

Książkę tę dedykujemy, jako wyraz uznania i podziwu, duchowi, który mieszka w komputerze.

„Myślę, że jest niezwykle ważne, aby nam — informatykom — programowanie ciągle sprawiało przyjemność. Na początku była to świetna zabawa. Oczywiście zdarzało się, że klienci byli nabijani w butelkę, ale z czasem zaczęliśmy poważnie traktować ich narzekania. Zaczęliśmy nawet uważać, że jesteśmy naprawdę odpowiedzialni za udane, bezbłędne, wręcz doskonałe zastosowanie komputerów. Nie sądzę, żeby tak było faktycznie. Myślę, że jesteśmy odpowiedzialni za poszerzanie zakresu ich zastosowań, badanie nowych kierunków i zapewnianie ludziom dobrej zabawy. Mam nadzieję, że zajmowanie się informatyką zawsze będzie sprawiać radość. Przede wszystkim jednak mam nadzieję, że nie staniemy się misjonarzami. Nie uważajcie się za sprzedawców Biblii. I tak jest ich już zbyt dużo na świecie. Tego, co wiecie o informatyce, inni ludzie też się nauczą. Nie sądzicie, że klucz do udanego zastosowania komputerów jest tylko w Waszych rękach. To, czym dysponujecie, jak mniemam i mam nadzieję, to inteligencja — zdolność dostrzegania w komputerze czegoś więcej niż tego, czym był, gdy go zobaczyliście po raz pierwszy, zdolność uczynienia z niego czegoś więcej”.

Alan J. Perlis (1 kwietnia 1922 r. – 7 lutego 1990 r.)

# Spis treści

Słowo wstępne	xiii
Przedmowa do drugiego wydania	xix
Przedmowa do pierwszego wydania	xxi
Podziękowania	xxv
<b>1. Budowanie abstrakcji za pomocą procedur</b>	<b>1</b>
1.1. Elementy programowania	4
1.1.1. Wyrażenia	5
1.1.2. Nazwy i środowisko	7
1.1.3. Obliczanie wartości kombinacji	8
1.1.4. Procedury złożone	11
1.1.5. Podstawieniowy model stosowania procedur	13
1.1.6. Wyrażenia warunkowe i predykaty	16
1.1.7. Przykład: pierwiastkowanie metodą Newtona	21
1.1.8. Procedury jako abstrakcyjne czarne skrzynki	25
1.2. Procedury i procesy przez nie generowane	30
1.2.1. Liniowa rekursja i iteracja	31
1.2.2. Rekursja drzewiasta	36
1.2.3. Rzędy wielkości	41
1.2.4. Potęgowanie	43
1.2.5. Największe wspólne dzielniki	47
1.2.6. Przykład: testowanie pierwszości	48
1.3. Formułowanie abstrakcji za pomocą procedur wyższych rzędów	55
1.3.1. Procedury jako argumenty	56
1.3.2. Tworzenie procedur za pomocą lambda-abstrakcji	60
1.3.3. Procedury jako metody ogólne	65
1.3.4. Procedury jako wyniki	71

---

<b>2. Budowanie abstrakcji za pomocą danych</b>	<b>79</b>
2.1. Wprowadzenie do abstrakcji danych	82
2.1.1. Przykład: operacje arytmetyczne na liczbach wymiernych	83
2.1.2. Bariery abstrakcji	87
2.1.3. Co rozumiemy przez dane?	90
2.1.4. Rozszerzone ćwiczenie: arytmetyka przedziałów	93
2.2. Dane hierarchiczne i własność domknięcia	97
2.2.1. Reprezentowanie ciągów	98
2.2.2. Hierarchiczne struktury danych	107
2.2.3. Ciągi jako konwencjonalne interfejsy	112
2.2.4. Przykład: język graficzny	126
2.3. Dane symboliczne	140
2.3.1. Cytowanie	141
2.3.2. Przykład: różniczkowanie symboliczne	144
2.3.3. Przykład: reprezentowanie zbiorów	150
2.3.4. Przykład: drzewa kodów Huffmana	159
2.4. Wielorakie reprezentacje danych abstrakcyjnych	167
2.4.1. Reprezentacje liczb zespolonych	169
2.4.2. Dane ze znacznikami	173
2.4.3. Programowanie sterowane danymi i addytywność	177
2.5. Systemy z operacjami ogólnymi	185
2.5.1. Ogólne operacje arytmetyczne	186
2.5.2. Łączenie danych różnych typów	191
2.5.3. Przykład: algebra symboliczna	199
<b>3. Modularność, obiekty i stany</b>	<b>213</b>
3.1. Przypisanie i stan lokalny	214
3.1.1. Lokalne zmienne stanu	215
3.1.2. Korzyści z wprowadzenia przypisania	221
3.1.3. Koszty wprowadzenia przypisania	225
3.2. Środowiskowy model obliczeń	232
3.2.1. Reguły obliczania	233
3.2.2. Stosowanie prostych procedur	236
3.2.3. Ramki jako magazyny stanów lokalnych	239
3.2.4. Definicje wewnętrzne	243
3.3. Modelowanie z użyciem danych modyfikowalnych	246
3.3.1. Modyfikowalne struktury listowe	247
3.3.2. Reprezentowanie kolejek	256
3.3.3. Reprezentowanie tablic	261

---

3.3.4. Symulator układów cyfrowych	267
3.3.5. Propagacja więzów	279
3.4. Współbieżność — istotny jest czas	290
3.4.1. Natura czasu w systemach współbieżnych	292
3.4.2. Mechanizmy sterowania współbieżnością	296
3.5. Strumienie	309
3.5.1. Strumienie jako listy odroczone	311
3.5.2. Strumienie nieskończone	319
3.5.3. Zastosowanie paradygmatu strumieniowego	327
3.5.4. Strumienie i obliczenia odroczone	339
3.5.5. Modularność programów funkcyjnych i modularność obiektów	345
<b>4. Abstrakcja metajęzykowa</b>	<b>351</b>
4.1. Ewaluator metacyjkliczny	354
4.1.1. Jądro ewaluatora	356
4.1.2. Reprezentowanie wyrażeń	360
4.1.3. Struktury danych ewaluatora	368
4.1.4. Uruchamianie ewaluatora jako programu	372
4.1.5. Dane jako programy	376
4.1.6. Definicje wewnętrzne	379
4.1.7. Oddzielanie analizy składowej od wykonywania	385
4.2. Wariacje na temat języka Scheme — leniwe obliczanie	390
4.2.1. Normalna i stosowana kolejność obliczania	390
4.2.2. Interpreter z leniwym obliczaniem	392
4.2.3. Strumienie jako listy leniwe	400
4.3. Wariacje na temat języka Scheme — obliczenia niedeterministyczne	403
4.3.1. Amb i przeszukiwanie	405
4.3.2. Przykłady programów niedeterministycznych	409
4.3.3. Implementacja ewaluatora amb	418
4.4. Programowanie w logice	429
4.4.1. Dedukcyjne wyszukiwanie informacji	433
4.4.2. Jak działa system zapytań	445
4.4.3. Czy programowanie w logice to logika matematyczna?	454
4.4.4. Implementacja systemu zapytań	460
<b>5. Obliczenia z użyciem maszyn rejestrowych</b>	<b>483</b>
5.1. Projektowanie maszyn rejestrowych	484
5.1.1. Język opisu maszyn rejestrowych	487

5.1.2. Abstrakcja w projektach maszyn	491
5.1.3. Podprogramy	494
5.1.4. Implementacja rekursji za pomocą stosu	498
5.1.5. Podsumowanie instrukcji	504
5.2. Symulator maszyny rejestrowej	505
5.2.1. Model maszyny	507
5.2.2. Asembler	511
5.2.3. Generowanie procedur wykonawczych dla instrukcji	514
5.2.4. Monitorowanie wydajności maszyn	522
5.3. Przydzielanie pamięci i odśmiecanie	524
5.3.1. Pamięć jako wektory	525
5.3.2. Podtrzymywanie złudzenia pamięci nieskończonej	531
5.4. Evaluator z jawnie określonym sterowaniem	537
5.4.1. Trzon evaluatora z jawnie określonym sterowaniem	539
5.4.2. Obliczanie ciągów wyrażeń i rekursja ogonowa	545
5.4.3. Instrukcje warunkowe, przypisania i definicje	548
5.4.4. Uruchamianie evaluatora	551
5.5. Kompilacja	556
5.5.1. Struktura kompilatora	560
5.5.2. Kompilowanie wyrażeń	564
5.5.3. Kompilowanie kombinacji	571
5.5.4. Łączenie ciągów instrukcji	578
5.5.5. Przykład skompilowanego kodu	581
5.5.6. Adresowanie składniowe	590
5.5.7. Scalanie skompilowanego kodu i evaluatora	594
Literatura	603
Wykaz ćwiczeń	611
Skorowidz	613

# Słowo wstępne

Nauczyciele, generałowie, dietetycy, psychologowie i rodzice programują. Działania wojska, studentów i niektórych społeczności są programowane. Zmierzenie się z dużym problemem wymaga wielu programów, z których większość jest powoływaną do życia w trakcie tych zmagań. Programy te są specyficzne dla rozpatrywanego problemu. Żeby docenić programowanie jako działalność intelektualną samą w sobie, trzeba zająć się programowaniem komputerów. Trzeba czytać i pisać programy komputerowe — dużo programów. Nie ma wielkiego znaczenia, czego one dotyczą ani do jakich celów służą. Znaczenie ma to, jak sprawnie działają i jak gładko dopasowują się do innych programów w trakcie tworzenia jeszcze większych programów. Programista musi dążyć zarówno do doskonałości elementów składowych, jak i do odpowiedniości całego ich zestawu. Pisząc o „programach”, w niniejszej książce autorzy skupią się na tworzeniu, wykonywaniu i badaniu programów napisanych w dialekcie Lispu i przeznaczonych do wykonania na komputerze cyfrowym. Używając Lispu, ograniczamy nie tyle to, co możemy zaprogramować, ile notację stosowaną do opisywania naszych programów.

Poruszane w tej książce problemy mają związek z ludzkim umysłem, zbiogrami programów komputerowych i komputerem. Każdy program komputerowy jest powstałym w umyśle modelem rzeczywistego bądź myślowego procesu. Procesy te, wynikające z ludzkiego doświadczenia i przemyśleń, są ogromnie liczne, zawiłe w szczegółach i zawsze tylko częściowo zrozumiałe. Rzadko się zdarza, byśmy byli trwale zadowoleni ze sposobu ich modelowania za pomocą programów komputerowych. Choć nasze programy są starannie, ręcznie przygotowanymi oddzielnymi zestawami symboli, mozaikami powiązanych ze sobą funkcji, stale ewoluują. Zmieniamy je w miarę, jak nasze pojmowanie modelu się pogłębia, poszerza i uogólnia, aż model ostatecznie osiąga metastabilny stan w ramach jeszcze innego modelu, z którym się zmagamy. Radość towarzysząca programowaniu komputerów ma swoje źródło w ciągłym rozwijaniu zarówno w umyśle, jak i na komputerze mechanizmów wyrażanych w postaci programów oraz w coraz lepszym rozumieniu różnych zjawisk dzięki tym wła-

śnie programom. Jak sztuka odtwarza nasze marzenia, tak komputer spełnia je poprzez programy!

Mimo całej swojej siły komputer jest bardzo wymagający. Przeznaczone dla niego programy muszą być poprawne, a to, co chcemy powiedzieć, musi być wyrażone precyzyjnie w najdrobniejszych szczegółach. Jak w każdym innym działaniu symbolicznym, o poprawności programów przekonujemy się drogą rozumowania. Można określić semantykę Lispu (będącą, swoją drogą, kolejnym modelem), a jeśli można wyspecyfikować funkcje programu, powiedzmy w rachunku predykatów, to za pomocą metod dowodzenia logiki da się w akceptowalny sposób wykazać poprawność programu. Ponieważ, niestety, programy stają się coraz większe i coraz bardziej skomplikowane, a to zdarza się niemal zawsze, odpowiedniość, spójność i poprawność samych specyfikacji staje pod znakiem zapytania. W tej sytuacji więc pełne, formalne dowody poprawności rzadko towarzyszą dużym programom. Ponieważ duże programy powstają z małych, bardzo istotne jest opracowanie arsenalu standardowych struktur programistycznych, których poprawności jesteśmy pewni (nazywamy je idiomami), i łączenie ich w większe struktury za pomocą technik organizacyjnych o sprawdzonej wartości. Techniki te są obszernie omówione w niniejszej książce, a ich zrozumienie ma zasadnicze znaczenie w prometejskim przedsięwzięciu zwanym programowaniem. Odkrywanie i biegłe opanowanie skutecznych technik organizacyjnych, bardziej niż cokolwiek innego, przyspiesza rozwijanie zdolności tworzenia dużych, znaczących programów. I odwrotnie, ponieważ pisanie dużych programów jest bardzo pracochłonne, stanowi bodziec do wynajdywania nowych metod ograniczania funkcji i szczegółów, jakie trzeba umieścić w programie.

W odróżnieniu od programów komputery podlegają prawom fizyki. Jeśli mają działać szybko (zmiana stanu w kilka nanosekund), muszą przesyłać elektrony jedynie na niewielkie odległości (co najwyższej  $1\frac{1}{2}$  stopy\*). Ciepło wytwarzane przez dużą liczbę gęsto upakowanych urządzeń musi być odprowadzane. Sztuka inżynierska rozwinęła się znakomicie w wyniku zmagań między mnogością funkcji a gęstością upakowania elementów. W każdym razie sprzęt zawsze działa na niższym poziomie niż ten, który nas interesuje na potrzeby programowania. Procesy, które przekształcają nasze programy w Lispie na programy „maszynowe”, same stanowią abstrakcyjne modele, które programujemy. Badanie i tworzenie ich daje dobry wgląd w programy organizacyjne związane z programowaniem dowolnych modeli. Oczywiście w ten sposób można zamodelować sam komputer. Pomyślimy: działanie najmniejszego fizycznego elementu przełączającego opisują równania różniczkowe mechaniki kwantowej, który to model bada się przybliżonymi metodami obliczeniowymi.

---

\* Około 45 cm (przyp. tłum.).

mi, możliwymi do użycia dzięki programom komputerowym, wykonywanym przez komputery, zbudowane z ... !

Rozróżnienie wspomnianych wcześniej trzech obszarów nie jest jedynie wygodnym wybierkiem taktycznym. Choć, jak powiadają, wszystko i tak jest w naszej głowie, rozróżnienie takie ułatwia symboliczną wymianę informacji między tymi obszarami, których bogactwo, witalność i potencjał rozwojowy ustępują jedynie ewolucji biologicznej. W najlepszym przypadku związki między nimi są metastabilne. Komputery nie są nigdy wystarczająco duże ani wystarczająco szybkie. Każdy przełom w technologii sprzętu prowadzi do jeszcze większych przedsięwzięć programistycznych, nowych zasad organizacyjnych i wzbogacenia modeli abstrakcyjnych. Każdy czytelnik powinien co pewien czas zadać sobie pytanie: „Dokąd to wszystko prowadzi, dokąd?” — jednak nie nazbyt często, żeby nie stracić radości programowania na rzecz miłego, ale bolesnego zarazem, filozofowania.

Wśród programów, które piszemy, niektóre (ale zawsze zbyt mało) wykonują ściśle określone funkcje matematyczne, takie jak sortowanie, wyszukiwanie maksimum w ciągu liczb, badanie pierwszości liczb bądź obliczanie pierwiastków kwadratowych. Programy takie nazywamy algorytmami — sporo wiadomo na temat ich optymalnego zachowania, a zwłaszcza o dwóch podstawowych parametrach: czasie wykonania i wymaganej pamięci. Programista powinien dobrze przyswoić sobie algorytmy i idiomy. Choć w wypadku niektórych programów trudno podać dokładne specyfikacje, obowiązkiem programisty jest oszacowanie wydajności programów i ciągłe podejmowanie prób poprawiania jej.

Język Lisp przetrwał, będąc w użyciu przez prawie czterowiecze. Spośród używanych języków programowania tylko Fortran może się poszczycić dłuższą historią. Oba te języki zaspokajały potrzeby programistyczne w ważnych obszarach zastosowań — Fortran w obliczeniach naukowych i inżynierskich, a Lisp w sztucznej inteligencji. Te dwa obszary są nadal ważne, a działający w nich programiści są tak oddani tym dwóm językom, że Lisp i Fortran mogą być jeszcze długo w użyciu, przynajmniej przez kolejne czterowiecze.

Lisp się zmienia. Scheme — dialekt używany w niniejszej książce — rozwinął się z początkowego Lispu i różni się od niego pod wieloma ważnymi względami, wliczając w to statyczne zakresy wiązania zmiennych i dopuszczenie, aby wartościami funkcji były funkcje. W swojej strukturze semantycznej Scheme jest w podobnym stopniu bliski Algolowi 60 jak wcześniejszym Lispm. Algol 60 nigdy już nie będzie w powszechnym użyciu, lecz jego geny przetrwały w języku Scheme i Pascale. Trudno byłoby znaleźć dwa języki, którymi porozumiewałyby się dwie tak bardzo różniące się od siebie społeczności jak te skupione wokół tych dwóch języków. Pascal służy do budowy piramid — imponujących, zapierających dech, statycznych struktur tworzonych przez armie przesuwające ciężkie kamienne bloki na swoje miejsca. Lisp służy do budowy

organizmów — równie imponujących, zapierających dech, dynamicznych struktur konstruowanych przez oddziały, które umieszczają w odpowiednich miejscowościach zmieniające się miriady prostszych organizmów. W obu przypadkach użyte zasady organizacyjne są takie same — z wyjątkiem jednej niezwykle ważnej różnicy. Otóż funkcjonalność programów, która może być przenoszona na zewnątrz, pozostawiona programiście piszącemu w Lispie do jego uznania, o ponad rząd wielkości przewyższa tę, którą można spotkać w projektach pas- calowych. Programy lispowe rozdymają biblioteki, dołączając do nich funkcje, których użyteczność wykracza poza zastosowania, dla których powstały. Za taki wzrost użyteczności w dużym stopniu są odpowiedzialne listy — rdzenna struktura danych Lispu. Prostota struktury i naturalna stosowalność list znalazły odbicie w funkcjach, które są niezwykle ogólne. W Pascalu mnóstwo deklarowalnych struktur danych powoduje specjalizację funkcji, która hamuje i utrudnia ich doraźne współdziałanie. Lepiej mieć 100 funkcji, które działają na jednej strukturze danych, niż 10, które działają na 10 strukturach danych. W rezultacie piramida tych struktur musi stać niezmieniona przez wieki, a organizm musi się rozwijać lub zginąć.

Aby zilustrować tę różnicę, porównaj, Drogi Czytelniku, sposób przedstawienia materiału i ćwiczenia zawarte w tej książce ze sposobem przedstawienia materiału i ćwiczeniami zawartymi w dowolnym podręczniku wstęp do programowania opartym na Pascalu. Nie ludź się, że ta książka jest podręcznikiem zrozumiałym tylko dla studentów MIT, odpowiednim dla typu ludzi, których można tam spotkać. Jest dokładnie taka, jaka powinna być poważna książka o programowaniu w Lispie — bez względu na to, dla kogo jest przeznaczona i gdzie jest używana.

Zwrót uwagę, że jest to książka o programowaniu — w odróżnieniu od większości podręczników Lispu, używanych na zajęciach przygotowujących do pracy nad sztuczną inteligencją. Mimo wszystko w miarę rozrastania się rozważanych systemów najważniejsze problemy programistyczne pojawiające się w inżynierii oprogramowania i sztucznej inteligencji zaczynają się zlewać. Tłumaczy to rosnące zainteresowanie Lispe'm poza dziedziną sztucznej inteligencji.

Badania nad sztuczną inteligencją, jak można by się spodziewać na podstawie celów, jakie przed tą dziedziną stoją, są źródłem wielu istotnych problemów programistycznych. W innych społecznościach programistycznych taki nawał problemów daje początek nowym językom. W rzeczywistości w każdym bardzo dużym przedsięwzięciu programistycznym użyteczna zasada organizacyjna polega na sterowaniu i izolowaniu komunikacji wewnętrz modułów przez wprowadzanie nowych języków. W miarę naszego zbliżania się do granic systemu, gdzie najczęściej zachodzą interakcje z ludźmi, języki te zwykle stają się mniej prymitywne. W rezultacie systemy takie zawierają wielokrotnie powtórzone, złożone funkcje przetwarzania języka. Lisp ma tak prostą składnię

i semantykę, że analiza składniowa może być uważana za zadanie elementarne. Tak więc technika analizy składniowej nie odgrywa prawie żadnej roli w programach lispowych, a budowa procesorów języka rzadko stanowi przeszkodę w tempie wzrostu i zmian dużych systemów lispowych. Wreszcie, to właśnie ta prostota składni i semantyki sprawia, że wszyscy programiści piszący w Lispie odczuwają brzemień programowania w tym języku, ale też wolność. Żaden program w Lispie, dłuższy niż ledwie kilka wierszy, nie może być zapisany bez nafaszerowania go funkcjami pomocniczymi. Wymyśl i dopasuj; spasuj i ulepsz! Pijemy zdrowie programistów piszących w Lispie, którzy ujęli swoje myśli w mnóstwie zagnieźdzonych nawiasów.

Alan J. Perlis  
New Haven, Connecticut

# Przedmowa do drugiego wydania

Czy to możliwe, że oprogramowanie nie jest podobne do niczego innego, że jest z góry przeznaczone do wyrzucenia — że cała rzecz polega na tym, aby zawsze traktować je jak bańkę mydlaną?

Alan J. Perlis

Materiał przedstawiony w tej książce był podstawą wykładu ze wstępu do informatyki na MIT od 1980 r. Pierwsze jej wydanie ukazało się po czterech latach prowadzenia przez nas zajęć według tego materiału, a drugie — po dwunastu latach. Jest nam bardzo miło, że nasze dzieło spotkało się z szerokim przyjęciem i że zawarte w nim treści zostały uwzględnione w wielu innych podręcznikach. Widzieliśmy, jak nasi studenci korzystają z rozwiązań i programów zawartych w tej książce, po czym tworzą z nich rdzeń nowych systemów komputerowych i języków. W dosłownym rozumieniu starożytnej talmudycznej gry słów, nasi studenci stali się naszymi budowniczymi. Jesteśmy szczęśliwi, że mamy tak zdolnych studentów i tak znakomitych budowniczych.

Przygotowując obecne wydanie, wprowadziliśmy setki objaśnień, które podsunęło nam nasze nauczycielskie doświadczenie, oraz komentarzy zasugerowanych nam przez naszych kolegów z i spoza MIT. Zmieniliśmy konstrukcję większości ważniejszych systemów programowania przedstawionych w tej książce, wliczając w to ogólny system arytmetyczny, interpretery, symulator maszyn rejestrowych i kompilator. Przepisaliśmy także wszystkie przykładowe programy, aby mieć pewność, że będą one działać dla dowolnej implementacji języka Scheme, spełniającej standard IEEE [55].

W niniejszym wydaniu kładziemy nacisk na kilka nowych tematów. Najważniejszy z nich to główna rola, jaką odgrywają w modelach obliczeniowych różne podejście do czasu: stosowanie obiektów ze stanami, programowanie współbieżne, programowanie funkcyjne, leniwe obliczanie i programowanie niedeterministyczne. Dołączliśmy nowe podrozdziały poświęcone współbieżności i niedeterminizmowi. Staraliśmy się też uwzględnić ten wątek w całej treści książki.

Materiał przedstawiony w pierwszym wydaniu ściśle odpowiadał programowi jednosemestrальnych zajęć z naszego przedmiotu na MIT. Nowy materiał z drugiego wydania wykracza poza program jednego semestru, a zatem wykładowca sam musi rozważyć, co jest mu do zajęć potrzebne. My sami czasami pomijamy podrozdział dotyczący programowania w logice (podrozdział 4.4), studenci używają symulatora maszyn rejestrowych, ale nie omawiamy jego

implementacji (podrozdział 5.2), jak również tylko pobicieżnie przedstawiamy budowę kompilatora (podrozdział 5.5). Nadal jednak jest to intensywny wykład. Niektórzy wykładowcy mogą zechcieć prowadzić zajęcia według materiału zawartego tylko w pierwszych trzech rozdziałach, pozostawiając resztę na wykłady w kolejnych semestrach.

Na stronie [www-mitpress.mit.edu/sicp](http://www-mitpress.mit.edu/sicp) można znaleźć materiały pomocnicze dla czytelników niniejszej książki. Są to między innymi programy zawarte w książce, przykładowe zadania programistyczne, materiały dodatkowe oraz implementacje dialekta Scheme języka Lisp, które możnaściągnąć i zainstalować.

# Przedmowa do pierwszego wydania

Komputer jest jak skrzypce. Można sobie wyobrazić nowicjusza, który najpierw sięga po gramofon, a następnie po skrzypce. Te ostatnie, jak mówią, wydają okropne dźwięki. Taką argumentację słyszeliśmy od naszych humanistów i większości informatyków-teoretyków. Programy komputerowe są dobre, jak mówią, do konkretnych celów, ale nie są elastyczne. Również skrzypce lub maszyna do pisania nie są elastyczne — dopóki się nie nauczysz ich używać.

Marvin Minsky: „Dlaczego programowanie jest dobrym sposobem wyrażania nie do końca zrozumiałych i byle jak sformułowanych koncepcji”.

„Struktura i interpretacja programów komputerowych” to wstępny przedmiot z informatyki na Massachusetts Institute of Technology. Jest obowiązkowy dla wszystkich studentów MIT, którzy jako główny przedmiot studiują elektrotechnikę lub informatykę. Stanowi jedną czwartą „wspólnego trzonu programowego”, na który składają się również dwa przedmioty dotyczące układów i systemów liniowych, a także projektowanie systemów cyfrowych. W opracowaniu tego przedmiotu byliśmy zaangażowani od 1978 r., a od jesieni 1980 r. wykładaliśmy go w jego obecnej postaci dla 600 do 700 studentów każdego roku. Większość z tych studentów nie miała żadnego formalnego przygotowania z programowania, choć wielu bawiło się wcześniej trochę komputerami, a kilku miało już duże doświadczenie w programowaniu lub projektowaniu sprzętu.

Opracowany przez nas program tego wstępnu do informatyki odzwierciedla dwa główne cele. Po pierwsze, chcemy uświadomić naszym studentom, że język komputerowy jest nie tyle środkiem zmuszającym komputer do wykonywania operacji co raczej nowym formalnym środkiem wyrażania koncepcji dotyczących metodyki. Tak więc programy powinny być pisane z myślą o ludziach, którzy mają je czytać, a możliwość ich wykonywania przez maszyny powinna być traktowana jako coś dodatkowego. Po drugie, wierzymy, że istotny materiał, jaki powinien być poruszony w ramach przedmiotu na tym poziomie, to nie składnia konstrukcji konkretnego języka programowania ani sprytne algorytmy efektywnego obliczania pewnych funkcji, ani też matematyczna analiza algorytmów czy podstawy obliczeń, lecz raczej techniki stosowane do panowania nad złożonością intelektualną dużych systemów oprogramowania.

Zależy nam na tym, żeby studenci, którzy kończą ten przedmiot, mieli dobre wyczucie podstaw stylu i estetyki programowania. Powinni znać główne techni-

ki nadzorowania złożoności dużych systemów. Powinni być w stanie przeczytać nawet 50-stronicowy program, jeśli jest on napisany przykładnie. Powinni wieźć, czego nie czytać, a czego w danej chwili nie muszą rozumieć. Powinni czuć się pewnie, modyfikując jakiś program, przy czym powinni zachować ducha i styl twórcy pierwotnej wersji programu.

Umiejętności te nie dotyczą wyłącznie programowania komputerów. Techniki, których uczymy i po które sięgamy, są wspólne dla wszystkich projektów inżynierskich. Panujemy nad złożonością programów przez tworzenie abstrakcji, które, gdy trzeba, ukrywają szczegóły. Panujemy nad złożonością przez ustanawianie konwencjonalnych interfejsów, które umożliwiają tworzenie systemów przez łączenie standardowych, dobrze znanych fragmentów metodą wybierania ich z różnych źródeł i dopasowywania do siebie w celu zbudowania spójnej całości. Panujemy nad złożonością przez ustanawianie nowych języków do opisywania projektu, z których każdy uwypukla pewne aspekty projektu, a pomniejsza inne.

U podstaw naszego podejścia do tego przedmiotu kryje się przekonanie, że „informatyka” nie jest nauką i że jej znaczenie ma niewiele wspólnego z komputerami. Rewolucja komputerowa to rewolucja w naszym sposobie myślenia i w sposobie wyrażania naszych myśli. Istotę tej zmiany stanowi pojawienie się czegoś, co można by najlepiej określić mianem *epistemologii proceduralnej* — badań struktury wiedzy z imperatywnego punktu widzenia w odróżnieniu od bardziej deklaratywnego punktu widzenia, który przyjmują klasyczne nauki matematyczne. Matematyka dostarcza ram do precyzyjnego zajmowania się pojęciami typu „co to jest”. Programowanie dostarcza ram do precyzyjnego zajmowania się pojęciami typu „jak to zrobić”.

W trakcie naszych zajęć używamy dialekту języka programowania Lisp. Nigdy formalnie nie uczymy tego języka, ponieważ nie musimy. Po prostu używamy go, a studenci w ciągu kilku dni sami go podchwytyją. Jest to jedna z istotnych zalet języków w rodzaju Lispu — umożliwiają one tworzenie wyrażeń złożonych tylko na kilka sposobów i nie mają prawie żadnej struktury składniowej. Wszystkie formalne własności można omówić w ciągu godziny, podobnie jak reguły szachów. Po krótkim czasie zapominamy o składniowych szczegółach języka (ponieważ ich brak) i przechodzimy do prawdziwych zagadnień — wyjaśnienia, co chcemy obliczyć, jak rozbijemy problem na łatwiejsze do ogarnięcia części i jak będziemy pracować nad tymi częściami. Inna zaleta Lispu polega na tym, że wspomaga on (ale nie wymusza) więcej strategii, dająccych duże możliwości zapewniające modularną dekompozycję programów, niż jakikolwiek inny język, który znamy. Możemy budować abstrakcje proceduralne i danych, możemy używać funkcji wyższego rzędu do uchwycenia powtarzających się schematów użycia, możemy modelować lokalne stany za pomocą przypisań i modyfikowania danych, możemy łączyć części programu za pomocą strumieni i obliczeń odroczonych, a także możemy z łatwością im-

plementować języki osadzone. Wszystko to jest umieszczone w interakcyjnym środowisku wspaniale wspomagającym przyrostowe projektowanie programów, ich konstruowanie, testowanie i odpluskwianie. Dziękujemy wszystkim pokoleniom magików lispowych, poczynając od Johna McCarthy'ego, który stworzył doskonale narzędzie o niespotykanej mocy i elegancji.

Scheme, dialekt Lispu, którego używamy, to wynik prób połączenia mocy i elegancji Lispu i Algolu. Z Lispu mamy metalingwistyczną moc, która bierze się z prostej składni, jednolitą reprezentację programów jako obiektów danych oraz dane alokowane dynamicznie i odśmiecanie. Z Algolu pochodzą składniowe wiązanie zmiennych i struktura blokowa — dary pionierów projektowania języków programowania, którzy tworzyli komitet Algolu. Chcielibyśmy wymienić tu Johna Reynoldsa i Petera Landina za zgłębienie przez nich związku między rachunkiem lambda Churcha a strukturą języków programowania. Uznajemy również nasz dług wobec matematyków, którzy zbadali to terytorium na dziesięciolecia przed pojawiением się na scenie komputerów. Do tych pionierów należą: Alonzo Church, Barkley Rosser, Stephen Kleene i Haskell Curry.

# Podziękowania

Chcielibyśmy podziękować całej rzeszy ludzi, którzy pomogli nam przygotować tę książkę i program zajęć.

Nasz przedmiot jest w prostej linii intelektualnym potomkiem „6.231” — cudownego przedmiotu z lingwistyki i rachunku lambda, wykłanego na MIT w późnych latach sześćdziesiątych (XX w.) przez Jacka Wozencrafta i Arthura Evansa, Jr.

Dużo zawdzięczamy Robertowi Fano, który tak zmienił program wstępnego nauczania na MIT na kierunkach elektrotechnika i informatyka, aby uwydniczyć zasady projektowania technicznego. Był naszym przewodnikiem, gdy zaczynaliśmy to przedsięwzięcie, i sporządził pierwsze notatki do przedmiotu, na podstawie których powstała niniejsza książka.

Wiele zagadnień dotyczących stylu i estetyki programowania, których staramy się uczyć, zostało opracowanych razem z Guyem Lewisem Steele’em Jr., który z Geraldem Jayem Sussmanem współtworzył na początku język Scheme. David Turner, Peter Henderson, Dan Friedman, David Wise i Will Clinger nauczyli nas wielu technik stosowanych przez społeczność programowania funkcyjnego, które pojawiają się w tej książce.

Joel Moses nauczył nas strukturalizować duże systemy. Jego doświadczenie z systemem obliczeń symbolicznych Macsyma zaowocowało spostrzeżeniem, że należy unikać złożonego sterowania i skupić się na takim zorganizowaniu danych, aby odzwierciedlały rzeczywistą strukturę modelowanego świata.

Marvin Minsky i Seymour Papert ukształtowali wiele z naszych poglądów na temat programowania i jego miejsca w naszym życiu intelektualnym. To dzięki nim zrozumieliśmy, że obliczenia stanowią środek wyrazu służący do badania koncepcji, które inaczej byłyby zbyt złożone, aby móc precyzyjnie się nimi zajmować. Podkreślają oni, że możliwość pisania i modyfikowania programów przez studentów jest potężną formą wyrazu, w której badanie staje się naturalną czynnością.

Zgadzamy się również całkowicie z poglądem Alana Perlisa, że programowanie to świetna zabawa i że powinniśmy bardziej dbać o to, żeby sprawiało nam radość. Część tej radości pochodzi z obserwowania wielkich mistrzów

przy pracy. Mieliśmy szczęście uczyć się programowania u Billa Gospera i Richarda Greenblatta.

Trudno wymienić wszystkich, którzy przyczynili się do opracowania naszego programu zajęć. Dziękujemy wszystkim wykładowcom, nauczycielom retoryki i opiekunom, którzy pracowali z nami przez ostatnich piętnaście lat i spędzili wiele dodatkowych godzin nad naszym przedmiotem. Wdzięczni jesteśmy zwłaszcza Billowi Siebertowi, Albertowi Meyerowi, Joemu Stoyowi, Randy'emu Davisowi, Louisowi Braidzie, Ericowi Grimsonowi, Rodowi Brookowi, Lynn Stein i Peterowi Szolovitsowi. Chcielibyśmy szczególnie podkreślić wybitny wkład Franklyna Turbaka (aktualnie w Wellesley) w nauczanie; jego praca ze studentami młodszych lat wyznaczyła standard, do którego wszyscy możemy dążyć. Jesteśmy wdzięczni Jerry'emu Saltzerowi i Jimowi Millerowi za pomoc w zmaganiach z tajemnicami współbieżności oraz Peterowi Szolovitsowi i Davidowi McAllesterowi za ich wkład w przedstawienie obliczeń niedeterministycznych w rozdziale 4.

Wielu ludzi miało istotny udział w zaprezentowaniu tego materiału na innych uniwersytetach. Wśród osób, z którymi współpracowaliśmy ściślej, są: Jacob Katzenelson z Technion, Hardy Mayer z University of California w Irvine, Joe Stoy z Oxford University, Elisha Sacks z Purdue i Jan Komorowski z Norges Teknisk-Naturvitenskapelige Universitet. Jesteśmy niezwykle dumni z naszych kolegów z innych uniwersytetów, którzy otrzymali liczące się nagrody za nauczanie swoich adaptacji tego przedmiotu, a wśród nich z Kennetha Yipa z Yale, Briana Harveya z University of California w Berkeley i Dana Huttenlochera z Cornell.

Al Moyé zorganizował wykłady, które prowadziliśmy dla inżynierów z Hewletta-Packarda, oraz produkcję kaset video z tymi wykładami. Chcielibyśmy podziękować utalentowanym nauczycielom — w szczególności Jimowi Millerowi, Billowi Siebertowi i Mike'owi Eisenbergowi. Przygotowali oni kursy kształcenia uzupełniającego, podczas których wykorzystano te taśmy, i prowadzili te kursy na uniwersytetach i w zakładach przemysłowych na całym świecie.

Wiele osób z innych krajów włożyło mnóstwo pracy w przetłumaczenie pierwszego wydania. Michel Briand, Pierre Chamard i André Pic są twórcami wydania francuskiego; Susanne Daniels-Herold jest twórcą wydania niemieckiego, a Fumio Motoyoshi — wydania japońskiego. Nie wiemy, kto przyczynił się do wydania chińskiego, niemniej jednak czujemy się zaszczyceni, że wybrano naszą książkę do „nieautoryzowanego” przekładu.

Trudno wyliczyć wszystkich ludzi, którzy mieli swój fachowy udział w opracowaniu systemów języka Scheme używanych przez nas w celach szkoleniowych. Oprócz Guya Steele'a wśród głównych magików należy wymienić Chrisa Hansona, Joego Bowbeera, Jima Millera, Guillermo Rozasa i Stephena Adamsa. Inni, którzy poświęcili wiele swojego czasu, to Richard Stallman,

Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larrabee, George Carrette, Soma Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Todd Cass, Patrick O'Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtemanche, Henry M. Wu, Andrew Berlin i Ruth Shyu.

Mając na uwadze nie tylko implementację opracowaną na MIT, chcielibyśmy też podziękować wielu ludziom, którzy pracowali nad standardem IEEE języka Scheme, w tym Williamowi Clingerowi i Jonathanowi Reesowi, którzy zredagowali R<sup>4</sup>RS, oraz Chrisowi Haynesowi, Davidowi Bartleyowi, Chrisowi Hansonowi i Jimowi Millerowi, którzy przygotowali standard IEEE.

Dan Friedman był przez długi czas przywódcą społeczności scheme'owej. Szerza działalność tej społeczności wykracza poza kwestie związane z projektowaniem języka i obejmuje wprowadzenie istotnych innowacji edukacyjnych, takich jak program nauczania dla szkół średnich oparty na EdScheme opracowanym przez Schemer's Inc., a także cudowne książki Mike'a Eisenberga oraz Briana Harveya i Matthew Wrighta.

Jesteśmy wdzięczni tym, którzy dołożyli starań, aby z naszego tekstu zrobić książkę, a zwłaszcza Terry Ehling, Larry'emu Cohenowi i Paulowi Bethge'owi z MIT Press. Ella Mazel znalazła cudowny rysunek na okładkę. Jeśli chodzi o drugie wydanie, to szczególne wyrazy podziękowania kierujemy do Bernarda i Elli Mazelów za pomoc przy projektowaniu książki oraz do Davida Jonesa, nadzwyczajnego magika TeX-owego. Jesteśmy również dłużnikami tych czytelników, którzy służyli nam wnikliwymi uwagami do nowej wersji tekstu. Dziękujemy za to Jacobowi Katzenelsonowi, Hardy'emu Mayerowi, Jimowi Millerowi, a szczególnie Brianowi Harveyowi, który zrobił dla tej książki to co Julie dla jego książki *Simply Scheme*.

Na koniec chcielibyśmy wyrazić swoje uznanie wspierającym nas firmom, które przez lata zachęcały nas do tej pracy, a w tym firmie Hewlett-Packard, której pomoc była możliwa dzięki Irze Goldsteinowi i Joelowi Birnbaumowi, a także agencji DARPA, z ramienia której wspomagał nas Bob Kahn.

# 1

## Budowanie abstrakcji za pomocą procedur

Trzy są czynności, w których umysł wykonuje swą władzę nad ideami prostymi: 1. łączenie pewnej liczby idei prostych w jedną, złożoną; tak powstają wszystkie idee złożone.

2. zestawienie łączone dwu idei, prostych lub złożonych, i to takie zestawienie, aby można je było na raz ogarnąć wzrokiem, nie stapiając ich w jedno; na tej drodze dochodzi umysł do wszelkich swych idei stosunków. 3. oddzielanie lub odrywanie idei od wszelkich innych, jakie im towarzyszą faktycznie w świadomości; nazywa się to abstrahowaniem i tak powstają wszystkie idee ogólne.

John Locke, *Rozważania dotyczące rozumu ludzkiego* (1690)\*

Zbadamy, co kryje się pod pojęciem *proces obliczeniowy* (ang. *computational process*). Procesy obliczeniowe to abstrakcje zamieszkujące komputery. W trakcie swojego życia korzystają z innych abstrakcyjnych obiektów zwanych *danymi*. Życie procesu przebiega zgodnie z zestawem reguł zwany *programem*. Ludzie tworzą programy, aby sterować procesami. W rezultacie za pomocą naszych reguł ujarzmiamy moce drzemiące w komputerze.

Proces obliczeniowy przypomina ducha z krainy magii. Nie można go zobaczyć ani dotknąć. Jest całkowicie niematerialny, a jednak bardzo rzeczywisty. Może wykonywać prace intelektualne. Potrafi odpowiadać na pytania. Oddziaływało na otaczający nas świat poprzez wypłacanie pieniędzy w banku lub sterowanie ramieniem robota w fabryce. Programy, których używamy, aby okiełznać procesy, są jak zaklęcia czarnoksiężnika — składają się ze starannie dobranych symbolicznych wyrażeń sekretnego i ezoterycznego *języka programowania* opisującego zadania, które procesy mają wykonywać.

W sprawnie działającym komputerze proces obliczeniowy wykonuje się ścisłe według programu. Początkujący programiści, tak jak uczniowie czarnoksiężnika, muszą nauczyć się rozumieć i przewidywać skutki swoich zaklęć. Nawet drobne błędy w programach (nazywane zazwyczaj *pluskiami*) mogą mieć poważne i nieprzewidziane skutki.

Na szczęście nauka programowania jest znacznie mniej niebezpieczna niż nauka czarów, ponieważ duchy, z którymi mamy do czynienia, są przechowy-

---

\* Przekład Bolesław J. Gawecki, PWN, Warszawa 1955 (przyp. tłum.).

wane w bezpieczny i dogodny sposób. W świecie rzeczywistym programowanie wymaga jednak uwagi, wiedzy i doświadczenia. Mały błąd na przykład w programie projektowania wspomaganego komputerowo może doprowadzić do katastrofy — upadku samolotu, pęknięcia tamy lub samozniszczenia robota przemysłowego.

Mistrzowie inżynierii oprogramowania posiadli umiejętność takiego konstruowania programów, iż mogły być dostatecznie pewni, że procesy będące wynikiem działania tych programów będą przebiegać tak, jak trzeba. Potrafią zawsze uzmysłowić sobie zachowanie tworzonych systemów. Wiedzą, jaka powinna być struktura programu, żeby nieoczekiwane problemy nie prowadziły do katastrofalnych skutków. Potrafią swoje programy *odpluskwiąć* w razie pojawienia się jakichkolwiek kłopotów. Dobrze zaprojektowane systemy obliczeniowe, tak jak dobrze zaprojektowane samochody czy reaktory jądrowe, mają strukturę modularną, dzięki czemu ich części mogą być tworzone, zastępowane i odpluskowane niezależnie.

### Programowanie w Lispie

Do opisywania procesów potrzebujemy odpowiedniego języka i będziemy do tego celu używać języka programowania Lisp. Nasze codzienne myślenie zwykłe wyrażamy w języku naturalnym (takim jak język polski, angielski czy francuski), a określenia zjawisk ilościowych w notacji matematycznej. Podobnie, nasze proceduralne myślenie będziemy wyrażać w Lispie. Lisp powstał w późnych latach pięćdziesiątych XX w. jako formalizm służący do wnioskowania o zastosowaniach pewnego rodzaju wyrażeń logicznych, nazywanych *równaniami rekurencyjnymi* (ang. *recursion equations*)\*, do modelowania obliczeń. Język ten został stworzony przez Johna McCarthy'ego i opiera się na jego pracy zatytułowanej „Recursive Functions of Symbolic Expressions and Their Computation by Machine” [71].

Pomimo że Lisp wymyślono jako formalizm matematyczny, jest on praktycznym językiem programowania. *Interpreter* Lispu to maszyna, która wykonuje procesy opisane w tym języku. Pierwszy interpreter Lispu został zaimplementowany przez McCarthy'ego i jego współpracowników oraz studentów z Artificial Intelligence Group z MIT Research Laboratory of Electronics oraz z MIT Computation Center<sup>1</sup>. Lisp, którego nazwa jest skrótem od LISP Processing (przetwarzanie list), zaprojektowano tak, aby umożliwić manipulowanie symbolami w sposób przydatny przy rozwiązywaniu takich problemów programistycznych, jak symboliczne różniczkowanie i całkowanie wyrażeń al-

\* Zgodnie z terminologią przyjętą przez wydawcę rozróżniamy pojęcia *rekursji* i *rekurencji*. Rekursja dotyczy języków programowania i definicji *procedur rekurencyjnych*, a rekurencja dotyczy *równań rekurencyjnych* (przyp. tłum.).

<sup>1</sup> *Lisp 1 Programmer's Manual* ukazał się w 1960 r., a *Lisp 1.5 Programmer's Manual* [74] został wydany w 1962 r. Wczesna historia Lispu jest opisana w [73].

gebraicznych. Stworzono do tego celu nowe obiekty danych, znane jako atomy i listy, co było najbardziej uderzającą różnicą między Lispem a wszystkimi innymi językami z tego okresu.

Lisp nie był produktem zaplanowanej pracy zespołu projektowego. Rozwijał się nieformalnie, na zasadzie eksperymentu, w odpowiedzi na potrzeby użytkowników i pragmatyczne rozwiązymania implementacyjne. Nieformalny rozwój Lispu trwał przez lata, a grono jego twórców tradycyjnie powstrzymywało się od prób ogłoszenia jakiejkolwiek „oficjalnej” definicji języka. Rozwój ten, w połączeniu z elastycznością i elegancją początkowej koncepcji języka, przyczynił się do tego, że Lisp, który jest drugim najstarszym językiem będącym dzisiaj w powszechnym użyciu (jedynie Fortran jest starszy), mógł być ciągle dostosowywany do najnowocześniejszych trendów w konstruowaniu programów. I tak dzisiaj Lisp tworzy rodzinę dialektów, które choć współdzielą większość z pierwotnych cech Lispu, mogą w znaczący sposób różnić się od siebie. W tej książce używamy dialekту Lispu o nazwie Scheme<sup>2</sup>.

Lisp był początkowo bardzo nieefektywny w zastosowaniu do obliczeń numerycznych, przynajmniej w porównaniu z Fortranem, z powodu swojego eksperymentalnego charakteru i położenia w nim nacisku na manipulowanie symbolami. Po latach jednak powstały kompilatory Lispu tłumaczące programy na kod maszynowy, który może wykonywać obliczenia numeryczne z rozsądnią efektywnością. W specjalnych zastosowaniach Lisp był używany z dobrym skutkiem<sup>3</sup>. Chociaż nadal panuje opinia, że jest to język beznadziejnie nieefektywny, ma on teraz wiele zastosowań, w których efektywność nie jest najważniejsza. Jest on na przykład wybierany jako język powłok systemów operacyjnych oraz jako język rozszerzeń w edytورach tekstu i systemach projektowania wspomaganego komputerowo.

---

<sup>2</sup> Dwa dialekty Lispu, w których napisano w latach siedemdziesiątych XX w. największe programy, to MacLisp [78, 82], opracowany w MIT Project MAC, oraz Interlisp [103], opracowany w Bolt Beranek and Newman Inc. i Xerox Palo Alto Research Center. Portable Standard Lisp [46, 38] zaprojektowano tak, aby można go było łatwo przenosić na różne maszyny. MacLisp dał początek kilku poddialektom, takim jak Franz Lisp, opracowany w University of California w Berkeley, oraz Zetalisp [79], oparty na wyspecjalizowanym procesorze zaprojektowanym w MIT Artificial Intelligence Laboratory i przeznaczonym do efektywnego wykonywania programów w Lispie. Dialekt Lispu stosowany w tej książce — Scheme [95] — został wymyślony w 1975 r. przez Guya Lewisa Steele'a Jr. i Geralda Jaya Sussmana z MIT Artificial Intelligence Laboratory, a później ponownie zaimplementowany w MIT do użycia w celach edukacyjnych. Scheme stał się standardem IEEE w 1990 r. [55]. Common Lisp [93, 94] był rozwijany przez grono użytkowników Lispu w celu połączenia cech wcześniejszych dialektów Lispu i ustanowienia przemysłowego standardu Lispu. Common Lisp stał się standardem ANSI w 1994 r. [3].

<sup>3</sup> Jedno z takich specjalnych zastosowań miało przełomowe znaczenie w nauce — scałkowanie ruchu układu słonecznego, dokładniejsze od poprzednich wyników o prawie dwa rzędy wielkości, wykazało, że dynamika układu słonecznego jest chaotyczna. Obliczenie to było możliwe dzięki nowym algorytmom całkowania, wyspecjalizowanemu kompilatorowi i wyspecjalizowanemu komputerowi — wszystko to zaimplementowane przy użyciu narzędzi programistycznych napisanych w Lispie [1, 100].

Skoro Lisp nie należy do głównego nurtu języków, to dlaczego używamy go jako podstawy w naszych rozważaniach o programowaniu? Jest tak, gdyż język ten ma wyjątkowe właściwości. Dzięki nim można go uznać za doskonały środek do uczenia się ważnych konstrukcji programistycznych i struktur danych oraz wiązania ich ze wspierającymi je konstrukcjami językowymi. Najbardziej znacząca z tych właściwości polega na tym, że w Lispie opisy procesów, nazywane *procedurami* (ang. *procedures*), można traktować jak dane i operować na nich jak na danych. Znaczenie tej właściwości bierze się stąd, że istnieją potężne techniki konstruowania programów, w których wykorzystuje się możliwości zacierania tradycyjnego rozróżnienia między „pasywnymi” danymi i „aktywnymi” procesami. Jak się okaże, fakt, że Lisp jest tak elastyczny, jeśli chodzi o posługiwanie się procedurami jako danymi, sprawia, iż jest jednym z najdogodniejszych języków do badania takich technik. Możliwość reprezentowania procedur jako danych czyni też z Lispu znakomity język do pisania programów, które przetwarzają inne programy jako dane, takich jak interpretery i kompilatory wspomagające języki programowania. A poza tym wszystkim programowanie w Lispie to po prostu świetna zabawa.

## 1.1. Elementy programowania

Język programowania o dużej mocy to coś więcej niż tylko środek służący do instruowania komputera, jak wykonywać zadania. Język taki służy też jako ramy, w których umieszczały nasze pojęcia dotyczące procesów. Tak więc, gdy opisujemy język, powinniśmy zwrócić szczególną uwagę na możliwości, jakie on udostępnia do łączenia pojęć prostych w celu formowania z nich pojęć bardziej złożonych. Każdy język programowania o dużej sile wyrazu zawiera trzy mechanizmy:

- **wyrażenia pierwotne** reprezentujące najprostsze elementy, których język dotyczy;
- **środki łączenia**, dzięki którym elementy złożone są budowane z elementów prostszych;
- **środki abstrakcji**, dzięki którym elementy złożone mogą być nazywane i traktowane jako całość.

W programowaniu mamy do czynienia z dwoma rodzajami elementów: procedurami i danymi. (Dalej okaże się, że w rzeczywistości nie różnią się one od siebie tak bardzo). Mówiąc nieformalnie, dane to „materiał”, który chcemy przetworzyć, a procedury to opisy reguł przetwarzania danych. Tak więc każdy język programowania o dużej sile wyrazu powinien móc opisywać dane pierwotne i procedury pierwotne, a także powinien uwzględniać metody służące do łączenia i abstrakcji procedur oraz danych.

W rozdziale tym będziemy się zajmować tylko prostymi danymi liczbowymi, co pozwoli nam się skupić na regułach budowania procedur<sup>4</sup>. W kolejnych rozdziałach zobaczymy, że te same reguły umożliwiają również budowanie procedur operujących na danych złożonych.

### 1.1.1. Wyrażenia

Prosty sposób rozpoczęcia nauki programowania polega na przyjrzeniu się kilku typowym interakcjom między użytkownikiem a interpreterem języka Scheme będącym dialektem Lispu. Wyobraź sobie, że siedzisz przy terminalu komputera. Wpisujesz *wyrażenia*, a interpreter odpowiada, wyświetlając wyniki *obliczenia wartości* tych wyrażeń.

Jednym z rodzajów wyrażeń pierwotnych, które mógłbyś wpisać, są liczby. (Ścisłej, wyrażenie, które wpisujesz, składa się z cyfr reprezentujących liczbę w systemie dziesiętnym). Jeśli podasz Lispowi liczbę

486

interpreter odpowie, wypisując<sup>5</sup>

486

Wyrażenia reprezentujące liczby można łączyć z wyrażeniem reprezentującym procedurę pierwotną (taką jak `+` czy `*`), tworząc wyrażenie złożone reprezentujące zastosowanie procedury do tych liczb. Na przykład:

`(+ 137 349)`

486

`(- 1000 334)`

666

---

<sup>4</sup> Określanie liczb jako „danych prostych” jest bezczelnym wprowadzaniem w błąd. W rzeczywistości sposób traktowania liczb jest jednym z najbardziej zawiłych i najbardziej mylących aspektów każdego języka programowania. Oto kilka typowych związanych z tym kwestii. Niektóre systemy komputerowe rozróżniają *liczby całkowite*, takie jak 2, i *liczby rzeczywiste*, takie jak 2.71. Czy liczba rzeczywista 2.00 różni się od liczby całkowitej 2? Czy działania arytmetyczne na liczbach całkowitych są takie same jak działania na liczbach rzeczywistych? Czy 6 podzielone przez 2 równa się 3, czy 3.0? Jak duże liczby mogą być reprezentowane? Z dokładnością do ilu cyfr możemy reprezentować liczby? Czy zakres liczb całkowitych jest taki sam jak zakres liczb rzeczywistych? Oprócz tych pytań jest jeszcze wiele innych kwestii dotyczących błędów zaokrąglenia — tworzą one całą dziedzinę analizy numerycznej. Ponieważ w niniejszej książce skupiamy się raczej na konstruowaniu programów w dużej skali niż na metodach numerycznych, więc pominiemy te kwestie. W przykładach liczbowych zamieszczonych w tym rozdziale występuje typowe zaokrąglanie, które można zaobserwować przy zastosowaniu działań arytmetycznych o ograniczonej precyzyji do liczb zmiennopozycyjnych.

<sup>5</sup> W niniejszej książce, ilekroć chcemy rozróżnić dane wejściowe wpisywane przez użytkownika i odpowiedzi wypisywane przez interpreter, te drugie zapisujemy pismem pochyłym.

(\* 5 99)

495

(/ 10 5)

2

(+ 2.7 10)

12.7

Wyrażenia takie jak te, nazywane *kombinacjami* (ang. *combinations*), powstają poprzez ujęcie listy wyrażeń w nawiasy w celu oznaczenia zastosowania procedury. Położony najbardziej na lewo (skrajnie lewy) element listy jest nazywany *operatorem*, a pozostałe elementy są nazywane *argumentami*. Wartość kombinacji jest otrzymywana przez zastosowanie określonej za pomocą operatora procedury do wartości argumentów.

Konwencja, w której operator jest umieszczany po lewej stronie argumentów, jest znana jako *notacja prefiksowa* (ang. *prefix notation*). Może być ona z początku nieco myląca, gdyż znacząco się różni od zwyczajowej konwencji matematycznej. Notacja prefiksowa ma jednak kilka zalet. Pierwsza z nich polega na tym, że można tak przystosować procedurę, aby ta mogła mieć dowolną liczbę argumentów, jak w następujących przykładach:

(+ 21 35 12 7)

75

(\* 25 4 12)

1200

Nie powstają przy tym żadne niejednoznacznosci, ponieważ operator jest zawsze skrajnie lewym elementem listy, a cała kombinacja jest ujęta w nawiasy.

Druga zaleta notacji prefiksowej polega na tym, że daje się ona w łatwy sposób rozszerzać, umożliwiając *zagnieżdżanie* kombinacji; tzn. możemy mieć kombinacje, których elementy są też kombinacjami:

(+ (\* 3 5) (- 10 6))

19

Nie ma (zasadniczo) ograniczenia na głębokość takiego zagnieżdżania ani na ogólny stopień skomplikowania wyrażeń, których wartość może obliczać interpretera Lispu. To tylko nas, ludzi, wprawiają w zakłopotanie nawet stosunkowo proste wyrażenia, takie jak

(+ (\* 3 (+ (\* 2 4) (+ 3 5))) (+ (- 10 7) 6))

które interpreter z łatwością obliczyłby jako 57. Możemy sobie ułatwić zadanie, zapisując takie wyrażenie w postaci

```
(+ (* 3
      (+ (* 2 4)
          (+ 3 5)))
      (+ (- 10 7)
          6))
```

stosując metodę formatowania nazywaną drukowaniem strukturalnym (ang. *pretty-printing*), zgodnie z którą każda długa kombinacja jest zapisywana tak, że kolejne argumenty są ustawione jeden pod drugim. Powstające wcięcia w klarowny sposób ukazują strukturę wyrażenia<sup>6</sup>.

Nawet w przypadku skomplikowanych wyrażeń interpreter zawsze działa w ten sam sposób: wczytuje wyrażenie z terminalu, oblicza wartość wyrażenia i wypisuje wynik. O takim trybie postępowania często mówi się, że interpreter działa w *pętli wczytaj-oblicz-wypisz*. Zauważmy w szczególności, że nie musimy jawnie instruować interpretera, aby wypisał wartość wyrażenia<sup>7</sup>.

### 1.1.2. Nazwy i środowisko

Krytycznym aspektem języka programowania jest sposób, w jaki pozwala on na używanie nazw w celu odwoływania się do obiektów obliczeniowych. Mówimy, że nazwa identyfikuje *zmienną* (ang. *variable*), której *wartością* (ang. *value*) jest obiekt.

W języku Scheme (dialekcie Lispu) nadajemy rzeczom nazwy za pomocą `define`. Wprowadzenie

```
(define size 2)
```

powoduje, że interpreter wiąże z nazwą `size` wartość 2<sup>8</sup>. Gdy tylko nazwa `size` zostanie związana z wartością 2, możemy odwoływać się do tej wartości, podając jej nazwę:

<sup>6</sup> Systemy lispowe zwykle udostępniają użytkownikowi udogodnienia wspomagające formatowanie wyrażeń. Dwa szczególnie użyteczne z nich to automatyczne ustawianie odpowiedniej głębokości wcięcia przy przejściu do nowego wiersza oraz podświetlanie właściwego nawiasu otwierającego, gdy wprowadzimy odpowiadający mu nawias zamkujący.

<sup>7</sup> W Lispie obowiązuje zasada, że każde wyrażenie ma wartość. Ta zasada razem ze starą złą opinią o Lispie jako języku nieefektywnym posłużyły Alanowi Perlisowi do dowcipnego sparfrazowania Oscara Wilda: „Programiści piszący w Lispie znają wartość wszystkiego, ale nie znają ceny niczego”.

<sup>8</sup> W książce tej nie pokazujemy, w jaki sposób interpreter odpowiada na obliczenie definicji, gdyż w dużym stopniu zależy to od implementacji.

```

size
2

(* 5 size)
10

```

Oto dalsze przykłady użycia `define`:

```

(define pi 3.14159)
(define radius 10)
(* pi (* radius radius))
314.159

(define circumference (* 2 pi radius))
circumference
62.8318

```

Korzystanie z `define` to najprostszy w naszym języku sposób tworzenia abstrakcji, gdyż umożliwia używanie prostych nazw na określenie wyników operacji złożonych, takich jak `circumference` powyżej. Na ogół obiekty obliczeniowe mogą mieć bardzo złożoną strukturę i pamiętanie oraz powtarzanie ich szczegółów za każdym razem, gdy chcemy z nich skorzystać, byłoby bardzo uciążliwe. Istotnie, złożone programy są budowane poprzez konstruowanie krok po kroku coraz bardziej złożonych obiektów obliczeniowych. Interpreter czyni to stopniowe konstruowanie szczególnie dogodnym, ponieważ powiązania obiektów z nazwami mogą być tworzone stopniowo w kolejnych interakcjach. Ta cecha interpretera zachęca do tworzenia programów w sposób przyrostowy i w znacznej mierze odpowiada za fakt, że programy napisane w Lispie zwykle składają się z dużej liczby względnie prostych procedur.

Powinno być oczywiste, że możliwość kojarzenia wartości z symbolami i późniejsze ich wyszukiwanie oznaczają, że interpreter musi obsługiwać jakiś rodzaj pamięci śledzącej pary skojarzonych ze sobą nazw i obiektów. Pamięć ta jest nazywana *środowiskiem* (a dokładniej *środowiskiem globalnym* (ang. *global environment*), gdyż jak później zobaczymy, obliczenie może dotyczyć wielu różnych środowisk)<sup>9</sup>.

### 1.1.3. Obliczanie wartości kombinacji

Jednym z celów, jakie stawiamy sobie w tym rozdziale, jest wyodrębnienie zagadnień związanych myśleniem proceduralnym. Rozważmy tę kwestię na przykładzie reguł, zgodnie z którymi interpreter oblicza wartości kombinacji.

---

<sup>9</sup> W rozdziale 3 zobaczymy, że pojęcie środowiska jest istotne zarówno do zrozumienia sposobu działania interpretera, jak i do zaimplementowania go.

- Aby obliczyć wartość kombinacji, należy wykonać następujące czynności:
  1. Obliczyć wartości podwyrażeń kombinacji.
  2. Zastosować procedurę będącą wartością skrajnie lewego podwyrażenia (operatora) do argumentów będących wartościami pozostałych podwyrażeń.

Nawet ta prosta reguła ilustruje pewne ważne kwestie dotyczące procesów w ogólności. Najpierw zauważmy, że w celu obliczenia wartości kombinacji w pierwszym kroku trzeba przede wszystkim obliczyć wartość każdego elementu kombinacji. Tak więc reguła ta jest *rekurencyjna* (ang. *recursive*) z natury; tzn. jako jeden ze swych kroków zawiera konieczność wywołania samej siebie<sup>10</sup>.

Zauważmy, jak pojęcie rekursji (ang. *recursion*) pozwala zwięźle ująć to, co w przypadku głęboko zagnieżdżonych kombinacji byłoby uważane za całkiem skomplikowany proces. Na przykład obliczenie wartości

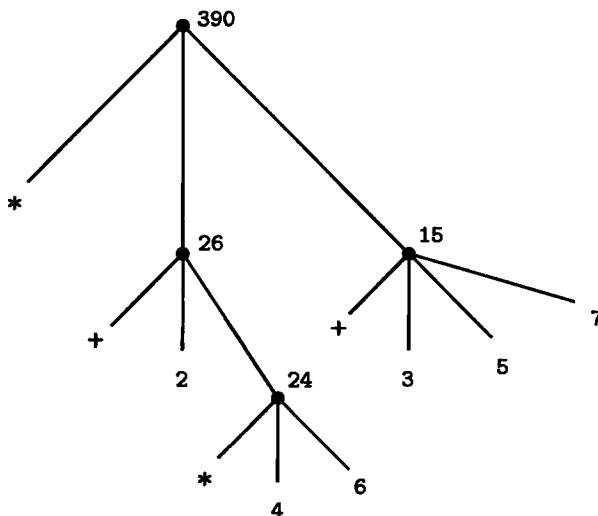
```
(* (+ (* 4 6))  
  (+ 3 5 7))
```

wymaga zastosowania reguły obliczania wartości do czterech różnych kombinacji. Proces obliczeniowy możemy zilustrować, przedstawiając kombinację w postaci drzewa, jak to widać na rys. 1.1. Każda kombinacja jest reprezentowana przez węzeł (wierzchołek) drzewa, z którego wychodzą gałęzie odpowiadające operatorowi i argumentom. Liście drzewa (tzn. węzły, z których nie wyrastają już żadne gałęzie) reprezentują albo operatory, albo liczby. Rozpatrując obliczanie wartości w kategoriach drzew, możemy sobie wyobrazić, że wartości operatorów i argumentów, poczynając od liści, przesuwają się w górę drzewa, łącząc się na coraz wyższych poziomach. Ogólnie biorąc, zobaczymy, że rekursja jest potężną techniką stosowaną przy używaniu hierarchicznych obiektów podobnych do drzew. W rzeczywistości obliczanie wartości na zasadzie „przesuwania się wartości w górę drzewa” jest przykładem ogólniejszej techniki znanej jako *akumulacja drzewiasta* (ang. *tree accumulation*).

Zauważmy dalej, że powtarzanie pierwszego kroku doprowadza nas do sytuacji, w której musimy obliczyć wartości już nie kombinacji, a wyrażeń pierwotnych, takich jak liczby, operatory wbudowane czy inne nazwy. W przypadku wyrażeń pierwotnych ustalamy następujące reguły:

- wartościami liczb są one same,

<sup>10</sup> Może się wydawać dziwne, że reguła obliczania wartości stanowi, iż w ramach pierwszego jej kroku powinniśmy obliczyć skrajnie lewy element kombinacji, skoro jak dotychczas może to być tylko operator (taki jak + czy -) reprezentujący wbudowaną procedurę pierwotną (taką jak dodawanie lub mnożenie). Dalej zobaczymy, że możliwość posługiwania się kombinacjami, których operatory same są wyrażeniami złożonymi, jest przydatna.



Rys. 1.1. Reprezentacja drzewiasta przedstawiająca wartości wszystkich podkombinacji

- wartością operatorów wbudowanych są sekwencje instrukcji (rozkazów) maszynowych wykonujących odpowiednie operacje,
- wartością innymi nazw są obiekty związane z tymi nazwami w środowisku.

Drugą z tych reguł możemy uważać za szczególny przypadek trzeciej reguły, przyjmując, że symbole takie jak + czy \* także należą do środowiska globalnego i że są z nimi związane sekwencje instrukcji maszynowych będące ich „wartościami”. Należy zwrócić uwagę na kluczową rolę środowiska w ustalaniu znaczenia symboli w wyrażenях. W języku interakcyjnym, takim jak Lisp, nie ma sensu mówić o wartości wyrażenia takiego jak (+ x 1), nie mówiąc nic o środowisku, które określiłoby znaczenie symbolu x (czy nawet symbolu +). Jak zobaczymy w rozdziale 3, ogólne pojęcie środowiska jako kontekstu obliczenia odgrywa ważną rolę w zrozumieniu wykonywania programów.

Zauważmy, że powyższe reguły obliczania wartości nie obejmują definicji. Na przykład obliczenie (define x 3) nie powoduje zastosowania define do dwóch argumentów, z których jeden jest wartością symbolu x, a drugi jest równy 3, gdyż celem define jest tu związywanie wartości z symbolem x. (Oznacza to, że (define x 3) nie jest kombinacją).

Takie wyjątki od ogólnej zasady obliczania wartości nazywamy *formami specjalnymi* (ang. *special forms*). Jak dotychczas define jest jedną formą specjalną, jaką spotkaliśmy, ale inne takie formy poznamy wkrótce. Każda forma specjalna ma własną regułę obliczania. Różne rodzaje wyrażeń (wraz z towarzyszącymi im regułami obliczania wartości) tworzą składnię języka

programowania. W porównaniu z większością innych języków programowania Lisp ma bardzo prostą składnię; oznacza to, że reguła obliczania wartości wyrażeń może być opisana za pomocą prostej, ogólnej reguły w połączeniu z wyspecjalizowanymi regułami dla kilku form specjalnych<sup>11</sup>.

#### 1.1.4. Procedury złożone

Wyróżniliśmy w Lispie kilka elementów, które muszą występować w każdym języku programowania o dużej sile wyrazu:

- Liczby i operacje arytmetyczne są pierwotnymi danymi i procedurami.
  - Zagnieżdżanie kombinacji udostępnia środki umożliwiające składanie operacji.
  - Definicje, wiążące nazwy z wartościami, zapewniają ograniczone środki abstrakcji.

Nauczymy się teraz *definiowania procedur* — dużo silniejszej techniki abstrakcji, dzięki której operacji złożonej można nadać nazwę, a następnie traktować ją jako całość.

Rozpoczniemy od zbadania, jak wyrazić zadanie „podnoszenia do kwadratu”. Możemy powiedzieć, że „aby podnieść coś do kwadratu, należy pomnożyć to coś przez siebie”. Możemy to zapisać w naszym języku jako

```
(define (square x) (* x x))
```

co możemy rozumieć następująco:

(define (square x) (\* x x))  
 ↑ ↑ ↑ ↑ ↑ ↑  
 Aby podnieść coś, należy pomnożyć to coś przez siebie.

Mamy tu do czynienia z procedurą złożoną (ang. *compound procedure*), której nadano nazwę **square**. Procedura ta reprezentuje operację mnożenia czegoś

<sup>11</sup> Specjalne formy składniowe, będące po prostu dogodnymi alternatywnymi strukturami powierzchniowymi dla czegoś, co może być zapisane w bardziej jednolity sposób, są czasem nazywane *lukrem syntaktycznym* (określenie ukute przez Petera Landina). Programiści piszący w Lispie, w porównaniu z użytkownikami innych języków, z reguły zwracają mniejszą uwagę na kwestie składniowe. (Dla porównania wystarczy przejrzeć dowolny podręcznik Pascala, aby zauważyc, jak duża jego część jest poświęcona opisowi składni). Ta pogarda dla składni wynika częściowo z elastyczności Lispu, dzięki której można łatwo zmieniać powierzchniowe elementy składni, a częściowo z obserwacji, że wiele „wygodnych” konstrukcji składniowych sprawia, iż język staje się mniej jednolity i w końcu, gdy programy stają się duże i skomplikowane, powodują więcej kłopotów, niż przynoszą korzyści. Mówiąc słowami Alana Perlisa: „Lukier syntaktyczny powoduje raka średnika”.

przez siebie. To coś, co ma być mnożone, ma nazwę lokalną `x`, która odgrywa taką samą rolę jak zaimek w języku naturalnym. Obliczenie definicji tworzy tę procedurę złożoną i wiąże ją z nazwą `square`<sup>12</sup>.

Ogólna postać definicji procedury jest następująca:

```
(define (<nazwa> <parametry formalne>) <treść>)
```

`<Nazwa>` to symbol, który w środowisku ma być związanym z definicją procedury<sup>13</sup>. `<Parametry formalne>` to nazwy używane w treści procedury na oznaczenie odpowiednich argumentów procedury. `<Treść>` procedury to wyrażenie, które po zastąpieniu parametrów formalnych rzeczywistymi argumentami (do których procedura jest zastosowana) określa wartość wyniku zastosowania procedury<sup>14</sup>. `<Nazwa>` i `<parametry formalne>` są ujęte razem w nawias okrągły, dokładnie tak jakby to było wywołanie definiowanej procedury.

Zdefiniowany procedurę `square`, możemy z niej korzystać:

```
(square 21)
```

```
441
```

```
(square (+ 2 5))
```

```
49
```

```
(square (square 3))
```

```
81
```

Możemy też użyć `square` jako elementu do budowy innych procedur. Na przykład  $x^2 + y^2$  może być wyrażone jako

```
(+ (square x) (square y))
```

Z łatwością możemy zdefiniować procedurę `sum-of-squares`, która mając dane jako argumenty dwie liczby, oblicza sumę ich kwadratów:

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

<sup>12</sup> Zauważmy, że mamy tu do czynienia z dwiema połączonymi czynnościami: tworzymy procedurę i nadajemy jej nazwę `square`. Możliwe, a co więcej ważne, jest rozdzielenie tych dwóch pojęć od siebie — tworzenie procedur bez nazywania ich oraz nazywanie procedur już utworzonych. W punkcie 1.3.2 zobaczymy, jak można to robić.

<sup>13</sup> Opisując w niniejszej książce ogólną składnię wyrażeń, stosujemy symbole pisane kursywą i ujęte w nawiasy ostre (np. `<nazwa>`) na oznaczenie w wyrażeniu „miejsc”, które należy wypełnić, chcąc użyć danego wyrażenia.

<sup>14</sup> Ogólniej rzecz biorąc, treść procedury może być sekwencją wyrażeń. W takim przypadku interpreter oblicza kolejno wszystkie te wyrażenia, a wynikiem zastosowania procedury jest wartość ostatniego z nich.

```
(sum-of-squares 3 4)  
25
```

Teraz możemy użyć `sum-of-squares` jako elementu do budowy dalszych procedur:

```
(define (f a)  
  (sum-of-squares (+ a 1) (* a 2)))  
  
(f 5)  
136
```

Procedur złożonych używamy dokładnie w ten sam sposób co procedur pierwotnych. Patrząc na powyższą definicję `sum-of-squares`, nie da się tak naprawdę stwierdzić, czy `square` jest procedurą wbudowaną, jak `+ i -`, czy też jest zdefiniowana jako procedura złożona.

### 1.1.5. Podstawieniowy model stosowania procedur

W trakcie obliczania wartości kombinacji, których operatorami są procedury złożone, interpreter postępuje podobnie jak w przypadku kombinacji, których operatorami są procedury pierwotne, co było opisane w punkcie 1.1.3. Oznacza to, że interpreter oblicza wartości elementów kombinacji, a następnie stosuje procedurę (będącą wartością operatora kombinacji) do argumentów (będących wartościami pozostałych elementów kombinacji).

Możemy założyć, że mechanizm stosowania procedur pierwotnych jest wbudowany w interpreter. W przypadku procedur złożonych proces zastosowania procedury przebiega następująco:

- Aby zastosować procedurę złożoną do argumentów, należy obliczyć wartość treści procedury z parametrami formalnymi zastąpionymi przez odpowiadające im argumenty.

W celu zilustrowania tego procesu obliczmy wartość kombinacji

```
(f 5)
```

gdzie `f` jest procedurą zdefiniowaną w punkcie 1.1.4. Zaczynamy od przypomnienia treści procedury `f`:

```
(sum-of-squares (+ a 1) (* a 2))
```

Następnie zastępujemy wszystkie wystąpienia parametru formalnego `a` argumentem 5:

```
(sum-of-squares (+ 5 1) (* 5 2))
```

W ten sposób problem sprowadza się do obliczenia wartości kombinacji operatora **sum-of-squares** i dwóch argumentów. Wykonanie tego wymaga rozwiązania trzech podproblemów. Musimy obliczyć wartość operatora, abytrzymać procedurę, która ma być zastosowana, oraz musimy obliczyć wartości argumentów. Wartość  $(+ 5 1)$  wynosi 6, a wartość  $(* 5 2)$  jest równa 10. Tak więc musimy zastosować procedurę **sum-of-squares** do argumentów 6 i 10. Argumenty te zastępują w treści procedury **sum-of-squares** parametry formalne  $x$  i  $y$ , redukując nasze wyrażenie do

$(+ (\text{square} 6) (\text{square} 10))$

Jeśli skorzystamy z definicji **square**, wyrażenie to sprowadzi się do

$(+ (* 6 6) (* 10 10))$

Wykonując mnożenia, otrzymujemy

$(+ 36 100)$

I w końcu uzyskujemy wynik

136

Opisany właśnie proces jest nazywany *podstawieniowym modelem* stosowania procedur. Może on być traktowany jako model wyznaczający „znaczenie” zastosowania procedury, przynajmniej na potrzeby procedur z tego rozdziału. Należy jednak podkreślić dwie rzeczy:

- Model podstawieniowy ma na celu ułatwienie nam wyobrażenia sobie zastosowania procedury, a nie udostępnienie nam opisu faktycznego działania interpretera. Typowe interpretery nie obliczają wyniku zastosowania procedury przez operowanie na treści procedury w celu podstawienia wartości argumentów w miejsce parametrów formalnych. W praktyce efekt „podstawienia” uzyskuje się, wykorzystując do reprezentowania wartości parametrów formalnych środowisko lokalne. Będziemy o tym mówić dokładniej w rozdziałach 3 i 4, gdy będziemy się szczegółowo zajmować implementacją interpretera.
- W książce tej będziemy przedstawiać coraz lepsze modele pracy interpretera, dochodząc w rozdziale 5 do pełnej implementacji interpretera i kompilatora. Model podstawieniowy jest tylko pierwszym z nich — dzięki niemu możemy zacząć myśleć o obliczaniu wartości w sposób formalny. Ogólnie mówiąc, gdy modelujemy zjawiska w nauce i technice, zaczynamy od uproszczonych i niepełnych modeli. W miarę coraz bardziej szczegółowego badania danego zjawiska uproszczone modele stają się nieadekwatne i muszą być zastąpione przez udoskonalone modele. Model podstawieniowy nie jest tu żadnym wyją-

kiem. W szczególności, gdy w rozdziale 3 będziemy się zajmować użyciem w procedurach „danych modyfikowalnych”, zobaczymy, że model podstawieniowy zawodzi i musi być zastąpiony przez bardziej skomplikowany model stosowania procedur<sup>15</sup>.

### **Stosowana a normalna kolejność obliczania**

Zgodnie z opisem obliczania wartości, podanym w punkcie 1.1.3, interpreter najpierw oblicza wartości operatora i argumentów, a dopiero potem stosuje otrzymaną procedurę do wartości argumentów. Nie jest to jedyny możliwy sposób obliczania wartości. W alternatywnym modelu wartości argumentów nie są obliczane, dopóki nie są potrzebne. Najpierw zastępujemy parametry formalne wyrażeniami podanymi jako argumenty aż do uzyskania wyrażenia zawierającego tylko operatory pierwotne, a następnie obliczamy wartość takiego wyrażenia. Obliczając w ten sposób wartość wyrażenia

(f 5)

otrzymalibyśmy kolejno następujące wyrażenia:

```
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)) )
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
```

które redukują się dalej do

```
(+ (* 6 6) (* 10 10))
(+ 36 100)
```

136

Otrzymany wynik jest taki sam jak w przypadku poprzedniego modelu obliczania wartości, ale proces obliczania jest inny. W szczególności dwukrotnie obliczana jest wartość wyrażeń  $(+ 5 1)$  i  $(* 5 2)$ , co odpowiada uproszczeniu wyrażenia

$(* x x)$

z  $x$  zastąpionym odpowiednio przez  $(+ 5 1)$  i  $(* 5 2)$ .

<sup>15</sup> Pomimo prostoty koncepcji dotyczącej podstawienia okazuje się, że jest ono niespodziewanie skomplikowane do zdefiniowania w ścisłych terminach matematycznych. Problem polega na możliwości pomylenia ze sobą nazw parametrów formalnych procedury i (potencjalnie takich samych) nazw występujących w wyrażenях, do których należy zastosować procedurę. Błędne definicje *podstawień* w literaturze dotyczącej logiki i semantyki programowania mają długą historię. Szczegółowe omówienie pojęcia podstawienia można znaleźć w [97].

Ta alternatywna metoda obliczania wartości przez „pełne rozwinięcie, a potem redukowanie” jest znana jako *normalna kolejność obliczania* (ang. *normal-order evaluation*) w odróżnieniu od metody „obliczania najpierw wartości argumentów, a następnie stosowania procedury”, używanej w rzeczywistości w interpreterach i nazywanej *stosowaną kolejnością obliczania* (ang. *applicative-order evaluation*). Można wykazać, że w wypadku zastosowań procedur, które mogą być modelowane za pomocą podstawień (dotyczy to wszystkich procedur przedstawionych w pierwszych dwóch rozdziałach tej książki) i dają poprawne wyniki, obliczanie wartości w normalnej i stosowanej kolejności daje te same wyniki. (Ćwiczenie 1.5 zawiera przykład procedury dającej „niepoprawne” wyniki, dla której obliczanie wartości w normalnej i stosowanej kolejności nie daje tych samych wyników).

W Lispie używana jest stosowana kolejność obliczania — częściowo ze względu na jej lepszą efektywność wynikającą stąd, że unikamy wielokrotnego obliczania wartości tych samych wyrażeń (jak było to widać powyżej na przykładzie  $(+ 5 1)$  i  $(* 5 2)$ ), a przede wszystkim dlatego, że obliczanie w normalnej kolejności staje się dużo bardziej skomplikowane, gdy wyjdziemy poza sferę procedur, które można modelować za pomocą podstawień. Normalna kolejność obliczania może jednakże być niezwykle wartościowym „narzędziem”, o czym się przekonamy w rozdziałach 3 i 4<sup>16</sup>.

### 1.1.6. Wyrażenia warunkowe i predykaty

Siła wyrazu klasy procedur, które nauczyliśmy się dotychczas definiować, jest bardzo ograniczona, gdyż nie umiemy w żaden sposób dokonywać wyboru i w zależności od niego wykonywać różne operacje. Nie potrafimy na przykład zdefiniować procedury obliczającej wartość bezwzględną liczby przez sprawdzenie, czy dana liczba jest dodatnia, ujemna, czy też równa zeru, i podjęcie w każdym z tych przypadków innych czynności, zgodnie ze wzorem

$$|x| = \begin{cases} x, & \text{jeśli } x > 0 \\ 0, & \text{jeśli } x = 0 \\ -x, & \text{jeśli } x < 0 \end{cases}$$

Konstrukcja taka jest nazywana *analizą przypadków* i w Lispie mamy dla niej specjalną formę zapisu. Oznacza się ją przez `cond` (co pochodzi od angielskiego słowa „conditional”, czyli „warunkowy”) i używa w taki oto sposób:

<sup>16</sup> W rozdziale 3 wprowadzimy *przetwarzanie strumieni* będące sposobem na radzenie sobie z najwyraźniej „nieskończonymi” strukturami danych przez wprowadzenie ograniczonej postaci normalnej kolejności obliczania. W podrozdziale 4.2 zmodyfikujemy interpreter języka Scheme tak, aby uzyskać wariant tego języka z normalną kolejnością obliczania.

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

Ogólna postać wyrażeń warunkowych cond w Lispie jest następująca:

```
(cond ((p1) (e1))
      ((p2) (e2))
      :
      ((pn) (en)))
```

Wyrażenie warunkowe składa się z symbolu cond, po którym następuje sekwencja ujętych w nawiasy par wyrażeń (*p*) (*e*) nazywanych *klauzulami* (ang. *clauses*). Pierwsze wyrażenie w każdej z tych par to *predykat* (ang. *predicate*) — tzn. wyrażenie, którego wartość jest interpretowana jako prawda albo fałsz<sup>17</sup>.

Wyrażenia warunkowe cond są obliczane następująco. Najpierw jest obliczana wartość predykatu *p*<sub>1</sub>. Jeśli wartość ta jest fałszem, to jest obliczana wartość *p*<sub>2</sub>. Jeśli *p*<sub>2</sub> jest również fałszem, to jest obliczana wartość *p*<sub>3</sub>. Proces ten jest kontynuowany, aż zostanie znaleziony predykat, którego wartość jest prawdą. Wówczas interpreter przyjmuje wartość odpowiadającego temu predykatowi *następnika klauzuli* (*e*) jako wartość całego wyrażenia warunkowego. Jeśli wartość żadnego z predykatów (*p*) nie jest prawdą, to wartość wyrażenia warunkowego jest nieokreślona.

Słowem „predykat” określamy zarówno procedury, których wynikiem jest prawda albo fałsz, jak i wyrażenia, których wartością jest prawda albo fałsz. Procedura *abs* obliczająca wartość bezwzględną korzysta z predykatów pierwotnych *>*, *<* i *=*<sup>18</sup>. W tych predykatach bierze się dwie liczby jako argumenty i sprawdza, czy liczba będąca pierwszym argumentem jest (odpowiednio) większa niż, mniejsza niż, czy równa liczbie będącej drugim argumentem, uzyskując w wyniku odpowiednio prawdę lub fałsz.

<sup>17</sup> „Interpretowana jako prawda albo fałsz” ma następujące znaczenie: W języku Scheme mamy dwie wyróżnione wartości reprezentowane przez stałe **#t** i **#f**. Kiedy interpreter sprawdza wartość predykatu, interpretuje **#f** jako fałsz, a każdą inną wartość jako prawdę. (Tak więc z logicznego punktu widzenia definiowanie stałej **#t** jest zbyteczne, aczkolwiek dogodne). W niniejszej książce używamy nazw **true** i **false** powiązanych odpowiednio z wartościami **#t** i **#f**.

<sup>18</sup> *Abs* korzysta również z operatora *-* (minus), który użyty tylko z jednym argumentem, jak w wyrażeniu *(- x)*, daje w wyniku liczbę przeciwną do danej.

Oto inny sposób zapisu procedury obliczającej wartość bezwzględną:

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

co w języku naturalnym można wyrazić tak: „Jeśli  $x$  jest mniejsze od zera, to wynikiem jest  $-x$ ; w przeciwnym razie wynikiem jest  $x$ ”. *Else* jest symbolem specjalnym, którego można użyć w miejscu  $\langle p \rangle$  w ostatniej klauzuli wyrażenia warunkowego *cond*. Powoduje on, że ilekroć wszystkie poprzedzające klauzule zostaną pominięte, wartością *cond* jest wartość odpowiadającego mu wyrażenia  $\langle e \rangle$ . W rzeczywistości dowolne wyrażenie, którego wartością jest zawsze prawda, może być użyte jako  $\langle p \rangle$ .

Oto jeszcze inny sposób zapisu procedury obliczającej wartość bezwzględną:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Używamy tu formy specjalnej *if* — ograniczonej formy wyrażenia warunkowego *cond* — z której możemy korzystać, gdy mamy do rozpatrzenia dokładnie dwa przypadki. Ogólna postać wyrażenia warunkowego *if* jest następująca:

```
(if <predykat> <następnik> <alternatywa>)
```

Obliczając wartość wyrażenia *if*, interpreter zaczyna od obliczenia wartości *<predykatu>*. Jeśli *<predykat>* ma wartość prawda, to interpreter jako wartość całego wyrażenia oblicza wartość *<następnika>*. W przeciwnym razie jako wartość wyrażenia obliczana jest wartość *<alternatywy>*<sup>19</sup>.

Oprócz relacji pierwotnych, takich jak  $<$ ,  $=$  i  $>$ , mamy do dyspozycji operacje logiczne umożliwiające budowanie predykatów złożonych. Trzy najczęściej używane z nich to:

- $(\text{and } \langle e_1 \rangle \dots \langle e_n \rangle)$

Interpreter oblicza kolejno od strony lewej do prawej wartości podwyrażeń  $\langle e \rangle$ . Jeśli którykolwiek z nich ma wartość fałsz, to kolejne podwyrażenia nie są obliczane, a wartością całego predykatu jest fałsz. Jeśli wszystkie podwyrażenia

---

<sup>19</sup> Niewielka różnica między *if* i *cond* polega na tym, że następnik  $\langle e \rangle$  każdej klauzuli *cond* może być sekwencją wyrażeń. Jeśli odpowiadający jej predykat  $\langle p \rangle$  ma wartość prawda, to obliczane są kolejno wartości wszystkich wyrażeń  $\langle e \rangle$  tworzących sekwencję i wartość ostatniego z nich jest wartością całego wyrażenia *cond*. Z kolei w wyrażeniu *if*  $\langle \text{następnik} \rangle$  i  $\langle \text{alternatywa} \rangle$  muszą być pojedynczymi wyrażeniami.

$\langle e \rangle$  mają wartość prawda, to wartością całego wyrażenia `and` jest wartość ostatniego z nich.

- `(or <e1> ... <en>)`

Interpreta oblicza kolejno od strony lewej do prawej wartości podwyrażeń  $\langle e \rangle$ . Jeśli którykolwiek z nich ma wartość prawda, to jego wartość jest wartością całego wyrażenia `or`, a kolejne podwyrażenia nie są obliczane. Jeśli wszystkie podwyrażenia  $\langle e \rangle$  mają wartość fałsz, to wartością całego wyrażenia `or` jest fałsz.

- `(not <e>)`

Wartość wyrażenia `not` jest prawdą, gdy wartość podwyrażenia  $\langle e \rangle$  jest fałszem, i na odwrotnie.

Zwrócmy uwagę, że `and` i `or` nie są procedurami, ale formami specjalnymi. Wynika to stąd, że nie wszystkie podwyrażenia będące ich argumentami muszą być obliczane. `Not` jest zwykłą procedurą.

Za przykład ich użycia może posłużyć warunek określający, że liczba  $x$  należy do przedziału  $5 < x < 10$ :

```
(and (> x 5) (< x 10))
```

Jako inny przykład niech nam posłuży predykat sprawdzający, czy jedna liczba jest większa lub równa drugiej, który możemy zapisać tak:

```
(define (>= x y)
  (or (> x y) (= x y)))
```

lub też tak:

```
(define (>= x y)
  (not (< x y)))
```

### Ćwiczenie 1.1

Poniżej jest podana sekwencja wyrażeń. Jaki jest wynik wypisywany przez interpreter w odpowiedzi na każde z nich? Załóż, że mają być one obliczane w takiej kolejności, w jakiej są podane.

```
10
(+ 5 3 4)
(- 9 1)
(/ 6 2)
(+ (* 2 4) (- 4 6))
(define a 3)
```

```

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))

```

### Ćwiczenie 1.2

Przedstaw w postaci prefiksowej następujące wyrażenie:

$$\frac{5 + 4 + \left(2 - \left(3 - \left(6 + \frac{4}{5}\right)\right)\right)}{3(6 - 2)(2 - 7)}$$

### Ćwiczenie 1.3

Zdefiniuj procedurę o trzech argumentach będących liczbami, której wynikiem jest suma kwadratów dwóch większych jej argumentów.

### Ćwiczenie 1.4

Zauważ, że nasz model obliczania wartości dopuszcza, aby operatorami były wyrażenia złożone. Korzystając z tej obserwacji, wyjaśnij działanie następującej procedury:

```

(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))

```

### Ćwiczenie 1.5

Ben Bajerbit wymyślił test pozwalający stwierdzić, czy interpreter, z którym mamy do czynienia, używa normalnej, czy też stosowanej kolejności obliczania. Zdefiniował on dwie procedury:

```

(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

```

a następnie wprowadził następujące wyrażenie:

```
(test 0 (p))
```

Co zaobserwowałby Ben w przypadku interpretera używającego stosowanej kolejności obliczania, a co w przypadku interpretera używającego normalnej kolejności obliczania? Odpowiedź uzasadnij. (Załóż, że reguła określająca obliczanie wartości formy specjalnej **if** nie zależy od tego, czy interpreter oblicza wartości w kolejności normalnej, czy też stosowanej: Najpierw jest obliczana wartość predykatu, od której zależy, czy dalej jest obliczana wartość następnika, czy alternatywy).

### 1.1.7. Przykład: pierwiastkowanie metodą Newtona

Przedstawione powyżej procedury są bardzo podobne do zwykłych funkcji matematycznych. Określają one wartość zależną od jednego lub więcej parametrów. Istnieje jednak istotna różnica między funkcjami matematycznymi a procedurami komputerowymi. Procedury muszą być efektywne.

Rozważmy to na przykładzie problemu obliczania pierwiastka kwadratowego. Funkcję pierwiastka kwadratowego możemy zdefiniować następująco:

$$\sqrt{x} = \text{takie } y, \text{ że } y \geq 0 \text{ i } y^2 = x$$

Jest to poprawna definicja funkcji matematycznej. Korzystając z niej, możemy stwierdzić, czy jedna liczba jest pierwiastkiem kwadratowym drugiej liczby, lub też wyciągać ogólne wnioski na temat pierwiastków kwadratowych. Jednakże definicja ta nie opisuje procedury. Faktycznie, nie mówi nam ona prawie nic o tym, jak znaleźć pierwiastek kwadratowy z danej liczby. Zapisanie tej definicji w pseudo-Lispie nic nie pomoże:

```
(define (sqrt x)
  (takie-że y (and (≥ y 0)
                    (= (square y) x))))
```

Opieramy się tu na nie istniejącej konstrukcji.

Różnica między funkcją a procedurą odzwierciedla ogólną różnicę między opisywaniem własności i czynności lub, jak to się czasem nazywa, różnicę między wiedzą deklaratywną i imperatywną. W matematyce interesują nas zwykle opisy deklaratywne (co to jest), podczas gdy w informatyce interesują nas zwykle opisy imperatywne (jak to zrobić)<sup>20</sup>.

<sup>20</sup> Opisy deklaratywne i imperatywne są blisko ze sobą związane, tak jak matematyka i informatyka. Stwierdzenie na przykład, że program daje „poprawne” wyniki jest stwierdzeniem deklaratywnym. Prowadzi się dużo badań mających na celu wypracowanie technik dowodzenia poprawności programów, a wiele związanych z tym trudności technicznych dotyczy uporania się z przejściem od konstrukcji imperatywnych (z których są zbudowane programy) do konstrukcji deklaratywnych (z których można korzystać w trakcie wnioskowania). W związku z tym

Jak oblicza się pierwiastki kwadratowe? Najbardziej powszechna jest metoda kolejnych przybliżeń Newtona, polegająca na tym, że ilekroć mamy przybliżoną wartość  $y$  pierwiastka kwadratowego liczby  $x$ , możemy w prosty sposób uzyskać lepsze przybliżenie pierwiastka: średnią arytmetyczną  $y$  i  $x/y$ <sup>21</sup>. Przykładowo, pierwiastek kwadratowy z 2 możemy obliczyć w następujący sposób. Przypuśćmy, że początkowo naszą przybliżoną wartością jest 1:

Wartość przybliżona	Iloraz	Wartość średnia
1	$\frac{2}{1} = 2$	$\frac{(2+1)}{2} = 1,5$
1,5	$\frac{2}{1,5} = 1,3333$	$\frac{(1,3333+1,5)}{2} = 1,4167$
1,4167	$\frac{2}{1,4167} = 1,4118$	$\frac{(1,4167+1,4118)}{2} = 1,4142$
1,4142	...	...

Kontynuując ten proces, uzyskujemy coraz lepsze przybliżenia pierwiastka kwadratowego.

Sformalizujmy ten proces, zapisując go w postaci procedur. Zaczynamy od liczby podpierwiastkowej (liczby, której pierwiastek próbujemy obliczyć) oraz pierwszej przybliżonej wartości pierwiastka. Jeśli przybliżenie jest dostatecznie dobre do naszych celów, to mamy gotowy wynik; jeśli nie, to musimy powtórzyć ten proces z lepszym przybliżeniem pierwiastka. Tę podstawową strategię możemy zapisać w postaci procedury:

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

---

ważnym obecnie kierunkiem działań w ramach konstruowania języków programowania jest zbadanie tzw. języków bardzo wysokiego poziomu, w których w rzeczywistości programuje się za pomocą konstrukcji deklaratywnych. Pomysł polega na tym, aby interpreterły były na tyle wyrafinowane, żeby na podstawie podanej przez programistę informacji „co to jest” same mogły automatycznie wygenerować informację „jak to zrobić”. Nie da się tego zrobić generalnie, ale w pewnych ważnych dziedzinach dokonano postępu. Wróćmy do tego tematu w rozdziale 4.

<sup>21</sup> Ten algorytm znajdowania pierwiastków kwadratowych jest szczególnym przypadkiem ogólniej metody Newtona znajdowania pierwiastków równań. Sam algorytm znajdowania pierwiastków kwadratowych został opracowany przez Herona z Aleksandrii w I w. n.e. W punkcie 1.3.4 zobaczymy, jak można zapisać ogólną metodę Newtona jako procedurę w Lispie.

Przybliżenie pierwiastka polepszamy, obliczając średnią z przybliżenia i liczby podpierwiastkowej podzielonej przez poprzednie przybliżenie pierwiastka:

```
(define (improve guess x)
  (average guess (/ x guess)))
```

gdzie

```
(define (average x y)
  (/ (+ x y) 2))
```

Musimy też powiedzieć, co to znaczy, że przybliżenie jest „dostatecznie dobre”. Poniższa definicja jest wystarczająca na potrzeby tego przykładu, choć nie jest w rzeczywistości najlepszym testem. (Zobacz ćwiczenie 1.7). Pomysł polega na przybliżaniu pierwiastka tak długo, aż kwadrat przybliżenia będzie się różnił od liczby podpierwiastkowej o mniej niż z góry ustalona wartość (w naszym przypadku 0,001)<sup>22</sup>:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

Na koniec musimy wiedzieć, jak zacząć. Możemy na przykład zawsze jako pierwsze przybliżenie pierwiastka kwadratowego dowolnej liczby przyjmować 1<sup>23</sup>:

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

Po wprowadzeniu do interpretera tych definicji możemy korzystać z `sqrt` jak z każdej innej procedury:

```
(sqrt 9)
3.00009155413138
```

<sup>22</sup> Zwykle będziemy predykatom nadawać nazwy kończące się znakiem zapytania, aby łatwiej pamiętać, że są to predykaty. Jest to jedynie konwencja stylistyczna. Dla interpretera znak zapytania jest tylko zwykłym znakiem.

<sup>23</sup> Zwróćmy uwagę, że nasze pierwsze przybliżenie zapisujemy raczej jako 1.0 niż jako 1. W wielu implementacjach Lispu nie powinno to robić żadnej różnicy. Jednakże MIT Scheme rozróżnia liczby całkowite i ułamki dziesiętne, a wynik dzielenia dwóch liczb całkowitych jest traktowany jak liczba wymierna, a nie ułamek dziesiętny. Na przykład 10 podzielone przez 6 jest równe  $\frac{5}{3}$ , podczas gdy 10.0 podzielone przez 6.0 jest równe 1.6666666666666667. (W punkcie 2.1.1 dowiemy się, jak można zaimplementować arytmetykę liczb wymiernych). Gdyby nasze początkowe przybliżenie pierwiastka kwadratowego było równe 1, a x było liczbą całkowitą, to wszystkie kolejno obliczane przybliżenia byłyby liczbami wymiernymi, a nie ułamkami dziesiętnymi. Operacje arytmetyczne, których jednym argumentem jest liczba wymierna, a drugim ułamek dziesiętny, zawsze dają w wyniku ułamek dziesiętny. Jeśli więc zaczniemy od 1.0 jako początkowego przybliżenia, to wszystkie kolejno obliczane przybliżenia będą ułamkami dziesiętnymi.

```
(sqrt (+ 100 37))
11.704699917758145

(sqrt (+ (sqrt 2) (sqrt 3)))
1.7739279023207892

(square (sqrt 1000))
1000.000369924366
```

Program `sqrt` pokazuje też, że prosty język proceduralny, jaki wprowadziliśmy do tej pory, jest wystarczający do pisania dowolnych czysto liczbowych programów, które można by zapisać, powiedzmy, w C czy Pascalu. Może się to wydawać zadziwiające, zważywszy, że nasz język nie zawiera żadnych konstrukcji iteracyjnych (pętli), sterujących komputerem tak, aby wielokrotnie powtarzał daną czynność. Z kolei `sqrt-iter` pokazuje, jak można uzyskać iterację bez żadnych specjalnych konstrukcji, z wyjątkiem zwykłego wywoływania procedur<sup>24</sup>.

### Ćwiczenie 1.6

Liz P. Haker nie rozumie, dlaczego `if` musi być formą specjalną: „Dlaczego nie mogę zdefiniować jej jako zwykłej procedury za pomocą `cond`?” Przyjaciółka Liz, Ewa Lu Ator, twierdzi, że rzeczywiście można tak zrobić, i definiuje nową wersję `if`:

```
(define (new-if predykat następnik alternatywa)
  (cond (predykat następnik)
        (else alternatywa)))
```

Ewa pokazuje Liz program:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Zachwycona Liz stosuje `new-if` w programie obliczającym pierwiastki kwadratowe:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))
```

Co się stanie, gdy Liz spróbuje użyć tego programu do obliczenia pierwiastka kwadratowego. Odpowiedź uzasadnij.

<sup>24</sup> Czytelnicy, którzy niepokoją się o kwestie związane z efektywnością zastosowania wywoływania procedur do zaimplementowania iteracji, powinni przeczytać uwagi dotyczące „rekursji ogonowej” podane w punkcie 1.2.1.

### Ćwiczenie 1.7

Funkcja `good-enough?` nie jest zbyt dobrym testem, gdy szukamy pierwiastków kwadratowych bardzo małych liczb. W prawdziwych komputerach operacje arytmetyczne są prawie zawsze wykonywane z ograniczoną precyzją. Czyni to nasz test nieodpowiednim również dla bardzo dużych liczb. Wyjaśnij powyższe stwierdzenia na przykładach obrazujących, jak funkcja `good-enough?` zawodzi dla małych i dużych liczb. Alternatywna metoda zaimplementowania tej funkcji polega na badaniu, jak w kolejnych krokach iteracji zmienia się przybliżona wartość pierwiastka, i zatrzymaniu iteracji, gdy ta zmiana jest tylko niewielkim ułamkiem przybliżonej wartości pierwiastka. Skonstruuj procedurę obliczającą pierwiastki kwadratowe, używającą takiego właśnie testu. Czy działa ona lepiej dla małych i dużych liczb?

### Ćwiczenie 1.8

Metoda Newtona obliczania pierwiastków sześciennych polega na tym, że jeżeli  $y$  jest przybliżoną wartością pierwiastka sześciennego z  $x$ , to

$$\frac{x/y^2 + 2y}{3}$$

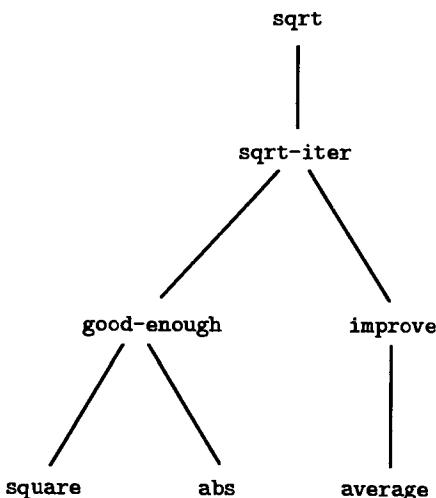
jest lepszym przybliżeniem. Korzystając z tego wzoru, zaimplementuj procedurę obliczającą pierwiastki sześciennne, analogiczną do procedury obliczającej pierwiastki kwadratowe. (W punkcie 1.3.4 dowiemy się, jak można zaimplementować ogólną metodę Newtona, będącą uogólnieniem procedur obliczających pierwiastki kwadratowe i sześciennne).

#### 1.1.8. Procedury jako abstrakcyjne czarne skrzynki

`Sqrt` było pierwszym przykładem procesu opisanego przez zestaw wzajemnie definiujących się procedur. Zauważmy, że definicja `sqrt-iter` jest *rekurencyjna*; oznacza to, że definicja procedury odwołuje się do siebie samej. Pomyśl definowania procedury za pomocą siebie samej może być niepokojący; może nie być jasne, czy taka „cykliczna” definicja ma jakikolwiek sens, a co dopiero czy poprawnie opisuje proces, jaki ma być wykonywany przez komputer. Zajmiemy się tym bliżej w punkcie 1.2. Najpierw jednak przyjrzyjmy się na przykładzie `sqrt` kilku innym ważnym kwestiom.

Zauważmy, że problem obliczania pierwiastków kwadratowych w naturalny sposób dzieli się na kilka podproblemów: jak stwierdzić, czy przybliżenie pierwiastka jest dostatecznie dobre; jak polepszyć przybliżenie itd. Każde z tych zadań jest realizowane przez oddzielną procedurę. Cały program `sqrt` może być przedstawiony jako zestaw procedur w sposób (widoczny na rys. 1.2) odzwierciedlający dekompozycję problemu na podproblemy.

Znaczenie dekompozycji nie polega tylko na tym, że dzielimy program na części. W końcu każdy duży program można podzielić na części — pierwszych dziesięć wierszy, następnych dziesięć wierszy, kolejnych dziesięć wierszy itd. Kluczowe jest to, że każda procedura realizuje pewne określone zadanie i stanowi moduł, którego można używać w definicjach innych procedur. Definiując



Rys. 1.2. Dekompozycja programu `sqrt` na procedury

na przykład procedurę `good-enough?` za pomocą procedury `square`, możemy tę ostatnią traktować jako „czarną skrzynkę”. Nie interesuje nas w danej chwili, *w jaki sposób* liczy ona wynik, a jedynie to, że podnosi liczby do kwadratu. Szczegóły dotyczące sposobu, w jaki liczby są podnoszone do kwadratu, mogą być odłożone na później. Faktycznie, dopóki interesuje nas tylko procedura `good-enough?`, `square` jest nie tyle procedurą co abstrakcją procedury, tzw. *abstrakcją proceduralną* (ang. *procedural abstraction*). Na tym poziomie abstrakcji każda procedura podnosząca liczby do kwadratu byłaby równie dobra.

Tak więc, biorąc pod uwagę jedynie wyniki, następujące dwie procedury podnoszenia do kwadratu są nie do odróżnienia. Każda z nich ma jeden argument będący liczbą i wynikiem każdej z nich jest kwadrat tej liczby<sup>25</sup>.

```

(define (square x) (* x x))

(define (square x)
  (exp (double (log x)))))

(define (double x) (+ x x))
  
```

Definicja procedury powinna umożliwiać ukrywanie szczegółów. Użytkownicy danej procedury mogli jej nie napisać sami, tylko dostać ją od innego

<sup>25</sup> Nie jest nawet jasne, która z tych procedur jest bardziej efektywną implementacją. Zależy to od dostępnego sprzętu. Istnieją maszyny, dla których „oczywista” implementacja jest mniej efektywna. Rozważ maszynę przechowującą w efektywny sposób szczegółowe tablice logarytmów i antylogarytmów.

programisty w postaci czarnej skrzynki. Aby korzystać z danej procedury, użytkownik nie musi wiedzieć, jak jest ona zaimplementowana.

### Nazwy lokalne

Jednym ze szczegółów implementacji procedury, który nie powinien być istotny dla jej użytkowników, są nazwy wybrane przez implementatora dla parametrów formalnych. Tak więc następujące dwie procedury nie powinny być rozróżnialne:

```
(define (square x) (* x x))  
(define (square y) (* y y))
```

Zasada ta — mówiąca, że znaczenie procedury nie powinno zależeć od nazw parametrów użytych przez jej autora — wydaje się na pierwszy rzut oka oczywista, jednak ma daleko idące konsekwencje. Najprostsza z nich polega na tym, że nazwy parametrów procedur muszą być lokalne w odniesieniu do treści procedury. Przykładowo, w procedurze pierwiastkowania, w definicji `good-enough?` używamy procedury `square`:

```
(define (good-enough? guess x)  
  (< (abs (- (square guess) x)) 0.001))
```

Intencją autora `good-enough?` było stwierdzenie, czy kwadrat pierwszego argumentu jest przybliżeniem, z określoną dokładnością, drugiego argumentu. Jak widać, autor `good-enough?` użył nazwy `guess` na oznaczenie pierwszego argumentu i nazwy `x` na oznaczenie drugiego argumentu. Argumentem `square` jest `guess`. Gdyby autor `square` użył `x` na oznaczenie tego argumentu (jak w definicji powyżej), to widać, że `x` w procedurze `good-enough?` musiałoby być innym `x` niż to w procedurze `square`. Uruchomienie procedury `square` nie może wpływać na wartość `x` używanego w `good-enough?`, ponieważ wartość ta może być potrzebna w `good-enough?` po zakończeniu procedury `square`.

Gdyby parametry nie były lokalne w odniesieniu do treści odpowiednich procedur, to parametr `x` procedury `square` mógłby zostać pomyłony z parametrem `x` procedury `good-enough?` i zachowanie procedury `good-enough?` zależało by od tego, której wersji procedury `square` użyliśmy. Zatem procedura `square` nie byłaby, jak tego chcieliśmy, czarną skrzynką.

Parametr formalny odgrywa w definicji procedury szczególną rolę, polegającą na tym, że nie ma znaczenia, jaką ma on nazwę. Taką nazwę określamy jako *zmienną związaną* (ang. *bound variable*) i mówimy, że definicja procedury *wiąże* swoje parametry formalne. Gdybyśmy konsekwentnie przemianowali zmienną związaną w całej definicji procedury, to znaczenie tej procedury nie

uległyby zmianie<sup>26</sup>. Jeśli zmienna nie jest związana, to mówimy, że jest *wolna* (ang. *free*). Zbiór wyrażeń, w obrębie których dane wiązanie definiuje nazwę, nazywamy *zakresem* (ang. *scope*) tej nazwy. W definicji procedury zakresem zmiennych związanych zadeklarowanych jako parametry formalne procedury jest treść procedury.

W powyższej definicji `good-enough?` zmienne `guess` i `x` są zmiennymi związanymi, ale `<`, `-`, `abs` i `square` są wolne. Znaczenie `good-enough?` nie powinno zależeć od nazw, jakie wybierzemy dla `guess` i `x` tak dugo, jak dugo są one różne i inne niż `<`, `-`, `abs` i `square`. (Gdybyśmy przemianowali `guess` na `abs`, powstałby błąd polegający na *zastonięciu* (ang. *capturing*) zmiennej `abs` — zamiast wolnej byłaby ona związana). Jednakże znaczenie procedury `good-enough?` nie jest niezależne od nazw jej zmiennych wolnych. Oczywiście zależy ono od tego, że `abs` jest nazwą procedury obliczającej wartość bezwzględną (co jest faktem zewnętrznym względem definicji procedury `good-enough?`). `Good-enough?` obliczy inną funkcję, gdy w jej definicji zastąpimy `abs` przez `cos`.

### Definicje wewnętrzne i struktura blokowa

Jak dotychczas poznaliśmy tylko jeden sposób izolowania nazw: parametry formalne procedury są lokalne w odniesieniu do treści tej procedury. Program obliczający pierwiastki kwadratowe ilustruje inny sposób, w jaki chcielibyśmy jeszcze kontrolować użycie nazw. Nasz program składa się z oddzielnych procedur:

```
(define (sqrt x)
  (sqrt-iter 1.0 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))
```

Problem polega tutaj na tym, że jedyną procedurą ważną dla użytkowników `sqrt` jest `sqrt`. Inne procedury (`sqrt-iter`, `good-enough?` i `improve`) tylko zaśmiecą im głowę. Nie mogą oni zdefiniować, jako części innego programu współpracującego z programem obliczającym pierwiastki kwadratowe, innej procedury o nazwie `good-enough?`, ponieważ `sqrt` jej potrzebuje.

<sup>26</sup> W rzeczywistości pojęcie konsekwentnego przemianowania jest subtelne i trudne do formalnego zdefiniowania. Nawet sławni logicy popełniali tu żenujące błędy.

Problem ten jest szczególnie poważny w przypadku dużych systemów tworzonych przez wielu programistów. Na przykład w dużej bibliotece procedur numerycznych wiele funkcji numerycznych jest obliczanych metodą kolejnych przybliżeń i mogą one mieć procedury pomocnicze o nazwach `good-enough?` i `improve`. Chcemy, by podprocedury stały się lokalne. W tym celu ukryjemy je w `sqrt`, żeby procedura `sqrt` mogła współistnieć z innymi procedurami wykorzystującymi metodę kolejnych przybliżeń, a każda z nich mogła mieć własną procedurę `good-enough?`. Aby było to możliwe, dopuszcamy, żeby procedura zawierała definicje wewnętrzne, które są lokalne względem tej procedury. W przypadku zadania obliczania pierwiastków kwadratowych możemy na przykład napisać następujący program:

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

Takie zagnieżdżenie definicji, nazywane *strukturą blokową* (ang. *block structure*), jest w zasadzie właściwym rozwiązaniem najbliższego problemu opakowywania nazw. Ale jest lepsze rozwiązanie. Oprócz uczynienia definicji procedur pomocniczych definicjami wewnętrznymi możemy je również uprosić. Ponieważ `x` jest zmienną związaną w definicji procedury `sqrt`, więc procedury `good-enough?`, `improve` i `sqrt-iter`, które są lokalne względem procedury `sqrt`, znajdują się w zakresie zmiennej `x`. Tak więc nie ma potrzeby przekazywania `x` w jawny sposób do każdej z tych procedur. Zamiast tego pozwalamy, aby `x` było w definicjach wewnętrznych zmienną wolną, jak jest to pokazane poniżej. Wówczas `x` otrzymuje wartość argumentu, z którym wywołano otaczającą procedurę `sqrt`. Taki sposób wiązania zmiennych nazywamy *wiązaniem składniowym* (ang. *lexical scoping*)<sup>27</sup>.

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess))))
```

<sup>27</sup> Wiązanie składniowe określa, że zmienne wolne w procedurze odnoszą się do definicji z procedury otaczającej; tzn. ich wartość jest określana przez środowisko, w którym dana procedura jest zdefiniowana. W rozdziale 3, gdy będziemy badali środowiska i szczegóły działania interpretora, zobaczymy dokładnie, jak funkcjonuje ten mechanizm.

```
(define (sqrt-iter guess)
  (if (good-enough? guess)
      guess
      (sqrt-iter (improve guess))))
(sqrt-iter 1.0))
```

Struktury blokowej będziemy często używać do rozbijania dużych programów na łatwe do ogarnięcia kawałki<sup>28</sup>. Pomyśl struktury blokowej pochodzi z języka programowania Algol 60. Pojawia się ona w większości zaawansowanych języków programowania i jest ważnym narzędziem pomagającym w organizacji budowy dużych programów.

## 1.2. Procedury i procesy przez nie generowane

Dotychczas rozważyliśmy następujące elementy programowania: użycie pierwotnych operacji arytmetycznych, składanie tych operacji i tworzenie abstrakcji operacji złożonych przez definiowanie ich jako procedur złożonych. Ale to za mało, żebyśmy mogli powiedzieć, iż wiemy już, jak należy programować. Jesteśmy w sytuacji analogicznej do tej, w której jest ktoś, kto wie, jak poruszają się figury w szachach, ale nie wie nic o typowych otwarciach, taktyce i strategii. Tak jak początkujący szachista, nie znamy podstawowych sposobów postępowania w naszej dziedzinie. Brak nam wiedzy o tym, które ruchy warto wykonać (które procedury są warte zdefiniowania). Brak nam też doświadczenia, aby przewidzieć skutki wykonania ruchu (wykonania procedury).

Umiejętność wyobrażania sobie skutków rozważanych akcji jest czynnikiem decydującym o tym, czy zostanie się wytrawnym programistą, tak samo jak w każdej innej syntetycznej i twórczej działalności. Aby zostać na przykład wytrawnym fotografem, trzeba się nauczyć, jak należy patrzeć na fotografowaną scenę, i trzeba wiedzieć, jaka powinna być jasność każdego fragmentu obiektu na zdjęciu dla każdych możliwych warunków ekspozycji i obróbki. Tylko wówczas, dobierając przyczyny do zamierzonych skutków, można tak zaplanować skadrowanie zdjęcia, oświetlenie, naświetlenie i obróbkę filmu, aby uzyskać pożądane efekty. Tak samo jest z programowaniem, gdy planujemy, jakie czynności ma wykonać proces, i sterujemy nim za pomocą programu. Aby zostać ekspertem, trzeba nauczyć się wyobrażać sobie procesy generowane przez różne rodzaje procedur. Tylko wtedy, kiedy osiągniemy takie umiejętności, będziemy mogli nauczyć się w niezawodny sposób tworzyć programy wykazujące pożądane zachowanie.

Procedura to wzorzec, zgodnie z którym następuje *lokalny rozwój* (ang. *local evolution*) procesu obliczeniowego. Precyzuje ona, jak w każdym kolejnym etapie procesu jest wykorzystywany poprzedni etap. Chcielibyśmy móc stwier-

<sup>28</sup> Definicje lokalne muszą występować na początku treści procedury. Producent nie odpowiada za skutki uruchamiania programów, w których definicje przeplatają się z użyciami.

dzać fakty dotyczące ogólnego, lub inaczej *globalnego*, zachowania procesów, których rozwój lokalny jest określony przez procedury. Jest to, ogólnie rzecz biorąc, bardzo trudne zadanie; możemy jednak przynajmniej spróbować opisać pewne typowe wzorce rozwoju procesu.

W tym podrozdziale rozpatrzymy pewne typowe „kształty” procesów generowanych przez proste procedury. Zbadamy również tempo, w jakim procesy te zużywają ważne zasoby obliczeniowe, takie jak czas i pamięć. Procedury, którymi się zajmiemy, są bardzo proste. Ich rola jest taka jak rola obrazów kontrolnych w fotografii: są to raczej nadmiernie uproszczone, prototypowe wzorce, a nie przykłady będące same w sobie praktycznymi programami.

### 1.2.1. Liniowa rekursja i iteracja

Zaczniemy od rozważenia następująco zdefiniowanej funkcji silnia:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1$$

Jest wiele sposobów obliczania silni. Jeden z nich polega na wykorzystaniu obserwacji, że  $n!$  jest równe  $n$  razy  $(n - 1)!$  dla każdej dodatniej liczby całkowitej  $n$ :

$$n! = n \cdot [(n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!$$

Tak więc możemy obliczyć  $n!$ , obliczając najpierw  $(n - 1)!$  i mnożąc wynik przez  $n$ . Jeśli dodamy do tego warunek mówiący, że  $1!$  jest równe 1, to powyższą obserwację możemy przedstawić w postaci takiej procedury:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Możemy użyć modelu podstawieniowego opisanego w punkcie 1.1.5 i zobaczyć tę procedurę w akcji — jak oblicza  $6!$ . Obliczenie to jest pokazane na rys. 1.3.

Spójrzmy na obliczanie silni z innej perspektywy. Możemy podać regułę obliczania silni, stwierdzając, że najpierw należy pomnożyć 1 przez 2, następnie uzyskany iloczyn pomnożyć przez 3, następnie przez 4 itd. aż do  $n$ . Mówiąc bardziej formalnie, przechowujemy bieżący iloczyn wraz z licznikiem zwiększającym wartość od 1 do  $n$ . Obliczenie możemy opisać za pomocą poniższych reguł określających, że licznik i iloczyn, krok po kroku, w następujący sposób zmieniają jednocześnie swoje wartości:

iloczyn  $\leftarrow$  licznik  $\cdot$  iloczyn

licznik  $\leftarrow$  licznik + 1

Przyjmujemy, że  $n!$  jest równe wynikowi mnożenia, gdy licznik przekroczy  $n$ .

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1))))))
(* 6 (* 5 (* 4 (* 3 2))))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Rys. 1.3. Liniowy proces rekurencyjny obliczający  $6!$

Ponownie możemy wyrazić nasz opis w postaci procedury obliczającej silnię<sup>29</sup>:

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Jak poprzednio, możemy użyć modelu podstawieniowego do przedstawienia procesu obliczającego  $6!$ , jak to widać na rys. 1.4.

Porównajmy te dwa procesy. Z jednej strony wydaje się, że prawie się one nie różnią. Oba obliczają tę samą funkcję matematyczną określoną na tej samej dziedzinie i oba, aby obliczyć  $n!$ , wykonują liczbę kroków proporcjonalną do  $n$ . Co więcej, oba procesy wykonują te same sekwencje mnożeń, uzyskując te sa-

<sup>29</sup> W prawdziwym programie prawdopodobnie użylibyśmy struktury blokowej, wprowadzonej w poprzednim podrozdziale, w celu ukrycia definicji `fact-iter`:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1)))))
  (iter 1 1))
```

Nie zrobiliśmy tak, chcąc zminimalizować liczbę rzeczy, które trzeba jednocześnie wziąć pod uwagę.

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Rys. 1.4. Liniowy proces iteracyjny obliczający 6!

me sekwencje wyników częściowych. Z drugiej strony jednak, gdy porównamy „kształty” obu procesów, okaże się, że rozwijają się one zupełnie inaczej.

Przyjrzyjmy się pierwszemu procesowi. Model podstawieniowy odsłania nam kształt, który się najpierw rozszerza, a następnie zwęża, zgodnie ze strzałką na rys. 1.3. Rozszerzanie następuje w miarę, jak proces buduje łańcuch *odłożonych operacji* (ang. *deferred operations*) — tutaj łańcuch mnożeń. Zwężanie następuje wówczas, gdy operacje te są faktycznie wykonywane. Taki rodzaj procesu, charakteryzujący się łańcuchem odłożonych operacji, jest nazywany *procesem rekurencyjnym*. Wykonanie takiego procesu wymaga, aby interpreter śledził, które operacje pozostały do wykonania na później. W trakcie obliczania  $n!$  długość łańcucha odłożonych mnożeń, a co za tym idzie ilość informacji, które należy śledzić, rośnie liniowo wraz ze wzrostem  $n$  (tzn. jest proporcjonalna do  $n$ ), tak samo jak liczba wykonywanych kroków. Taki proces jest nazywany *liniowym procesem rekurencyjnym* (ang. *linear recursive process*).

W odróżnieniu od pierwszego „kształtu” drugiego procesu nie rozszerza się ani nie zwęża. W każdym kroku i dla każdego  $n$  jedyne, co musimy śledzić, to bieżące wartości zmiennych *product*, *counter* i *max-count*. Taki proces nazywamy *iteracyjnym*. Ogólnie mówiąc, proces iteracyjny to taki proces, którego stan może być opisany przez ustaloną liczbę *zmiennych stanu* (ang. *state variables*), którego zmiany stanów możemy opisać ustaloną regułą mówiącą, jak zmieniają się wartości zmiennych stanu, oraz którego zakończenie (opcjonalnie) możemy opisać za pomocą warunku końcowego. Przy obliczaniu  $n!$  liczba kroków rośnie liniowo wraz z  $n$ . Taki proces nazywamy *liniowym procesem iteracyjnym* (ang. *linear iterative process*).

Na różnice między tymi dwoma procesami możemy też spojrzeć inaczej. W przypadku procesu iteracyjnego w każdej chwili zmienne programu w pełni opisują stan procesu. Gdybyśmy wstrzymali obliczenia między dwoma krokami, to do ich wznowienia wystarczyłoby podać interpreterowi wartości trzech zmiennych występujących w programie. W przypadku procesu rekurencyj-

nego tak już nie jest. Mamy tutaj do czynienia z pewną „ukrytą” informacją, nie wynikającą ze zmiennych programu, utrzymywana przez interpreter i wskazującą „gdzie proces się znajduje” w trakcie obliczania łańcucha odłożonych operacji. Im dłuższy łańcuch, tym więcej informacji musi być pamiętanych<sup>30</sup>.

Porównując iterację i rekursję, musimy być ostrożni, aby nie pomylić pojęcia *procesu* rekurencyjnego z pojęciem *procedury* rekurencyjnej. Gdy mówimy, że procedura jest rekurencyjna, odnosimy się wtedy do faktu składniowego, polegającego na tym, że definicja procedury odwołuje się (bezpośrednio albo pośrednio) do siebie samej. Gdy zaś mówimy, że proces działa zgodnie z pewnym wzorcem — jest on, powiedzmy, liniowo rekurencyjny — wtedy mówimy o tym, jak proces się rozwija, a nie o składni zapisu procedury. Może nas niepokoić, że procedura rekurencyjna, taka jak `fact-iter`, generuje proces iteracyjny. Jednakże proces jest rzeczywiście iteracyjny: jego stan jest całkowicie wyznaczony przez jego trzy zmienne stanu, a interpreter w celu wykonania tego procesu musi śledzić tylko owe trzy zmienne.

Jeden z powodów, dla których rozróżnienie między procesami i procedurami może być mylące, wynika stąd, iż większość powszechnie używanych języków programowania (w tym Ada, Pascal i C) jest tak zaprojektowana, że interpretacja jakiekolwiek procedury rekurencyjnej wymaga tym więcej pamięci, im więcej występuje wywołań procedury, nawet jeśli opisywany przez nią proces jest w zasadzie iteracyjny. Dlatego też w językach tych możemy opisywać procesy iteracyjne tylko przy wykorzystaniu wyspecjalizowanych „konstrukcji pętli”, takich jak `do`, `repeat`, `until`, `for` i `while`. Implementacja języka Scheme, którą będziemy rozpatrywać w rozdziale 5, jest pozbawiona tej wady. Proces iteracyjny będzie się tam wykonywał w stałej pamięci, nawet jeśli proces ten jest opisany za pomocą procedury rekurencyjnej. Jeśli implementacja ma tę cechę, to mówimy wówczas o *rekursji ogonowej* (ang. *tail recursion*). Mając do dyspozycji rekursję ogonową, możemy wyrazić iterację za pomocą zwykłego mechanizmu wywoływanego procedur, a co za tym idzie specjalne konstrukcje iteracyjne są przydatne jedynie jako lukier syntaktyczny<sup>31</sup>.

---

<sup>30</sup> Gdy w rozdziale 5 będziemy omawiać implementację procedur na maszynie rejestrowej, zobaczymy, że każdy proces iteracyjny może być zrealizowany „sprzętowo” w postaci maszyny z ustalonym zestawem rejestrów i bez żadnej pamięci pomocniczej. Z kolei realizacja procesu rekurencyjnego wymaga maszyny wyposażonej w pomocniczą strukturę danych nazywaną *stosem* (ang. *stack*).

<sup>31</sup> Rekursja ogonowa długo była uważana za trik optymalizacyjny stosowany w komplikacji. Spójne podstawy semantyczne rekursji ogonowej zostały podane przez Carla Hewitta [50]. Wyjaśnił on rekursję ogonową, odwołując się do modelu obliczeń opartego na „przesyłaniu komunikatów”, który omówimy w rozdziale 3. Zainspirowani tym Gerald Jay Sussman i Guy Lewis Steele Jr. skonstruowali interpreter języka Scheme z rekursją ogonową [95]. Steele pokażał później, jak rekursja ogonowa wynika z naturalnego sposobu komplikowania wywołań

**Ćwiczenie 1.9**

Każda z następujących dwóch procedur definiuje metodę dodawania dwóch dodatnich liczb całkowitych za pomocą procedury `inc`, która zwiększa swój argument o 1, i procedury `dec`, która zmniejsza swój argument o 1.

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b)))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Używając modelu podstawieniowego, zilustruj procesy generowane przez każdą procedurę przy obliczaniu wartości  $(+ 4 5)$ . Czy są to procesy iteracyjne, czy też rekurencyjne?

**Ćwiczenie 1.10**

Następująca procedura oblicza funkcję matematyczną nazywaną funkcją Ackermann'a:

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                  (A x (- y 1))))))
```

Jakie wartości mają poniższe wyrażenia?

(A 1 10)

(A 2 4)

(A 3 3)

Rozważ następujące procedury, w których `A` jest procedurą zdefiniowaną powyżej:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Podaj zwięzłe matematyczne definicje funkcji obliczanych przez procedury `f`, `g` i `h` dla dodatnich całkowitych wartości  $n$ . Na przykład:  $(k n)$  oblicza  $5n^2$ .

---

procedur [92]. Standard IEEE języka Scheme wymaga, aby w implementacjach tego języka była stosowana rekursja ogonowa.

### 1.2.2. Rekursja drzewiasta

Innym powszechnie stosowanym wzorcem obliczeń jest *rozgałęzająca się rekursja* (*rekursja drzewiasta*; ang. *tree recursion*). Jako przykład rozważmy obliczanie ciągu liczb Fibonacciego, w którym każdy element jest sumą dwóch poprzednich:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Na ogół liczby Fibonacciego mogą być zdefiniowane następującym wzorem:

$$\text{Fib}(n) = \begin{cases} 0, & \text{jeśli } n = 0 \\ 1, & \text{jeśli } n = 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{w przeciwnym razie} \end{cases}$$

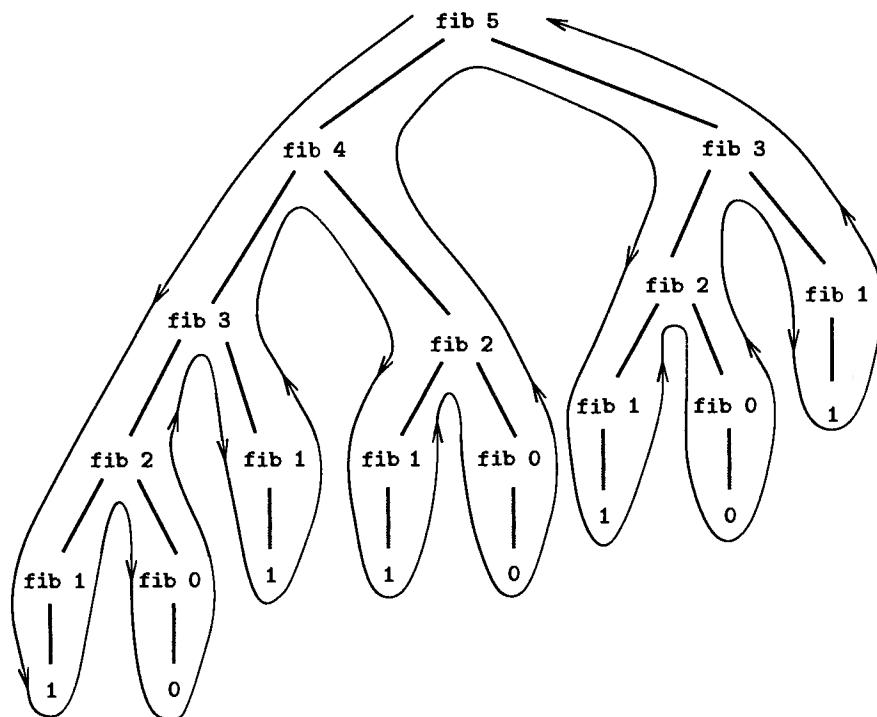
Możemy ten wzór od razu przetłumaczyć na rekurencyjną procedurę obliczającą liczby Fibonacciego:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2)))))))
```

Rozważmy wzorzec takiego obliczenia. Aby obliczyć `(fib 5)`, obliczamy `(fib 4)` i `(fib 3)`. Aby obliczyć `(fib 4)`, obliczamy `(fib 3)` i `(fib 2)`. Ogólnie mówiąc, rozwijający się proces ma kształt drzewa, co widać na rys. 1.5. Zauważmy, że na każdym poziomie (z wyjątkiem samego dołu) gałęzie rozdzielają się na dwoje; odzwierciedla to fakt, że procedura `fib` za każdym razem dwukrotnie wywołuje samą siebie.

Procedura ta jest pouczającym przykładem rozgałęzającej się rekursji, ale jest okropnym sposobem obliczania liczb Fibonacciego, gdyż wykonuje mnóstwo nadmiarowych (zewnętrznych) obliczeń. Patrząc na rys. 1.5, zauważymy, że obliczenie wartości `(fib 3)` — prawie połowa całej roboty — jest wykonywane dwukrotnie. W rzeczywistości można łatwo pokazać, że liczba wywołań `(fib 1)` i `(fib 0)` (zazwyczaj równa liczbie liści w powyższym drzewie obliczeń) wynosi dokładnie  $\text{Fib}(n + 1)$ . Aby uzmysłowić sobie, jak zły jest to rezultat, można pokazać, że wartość  $\text{Fib}(n)$  rośnie wykładniczo ze względu na  $n$ . Dokładniej (zob. ćwiczenie 1.13),  $\text{Fib}(n)$  jest liczbą całkowitą najbliższą  $\phi^n / \sqrt{5}$ , gdzie

$$\phi = \frac{(1 + \sqrt{5})}{2} \approx 1,6180$$



Rys. 1.5. Proces rozgałęziający się rekurencyjnie, generowany przy obliczaniu (fib 5)

jest złotym podziałem, spełniającym równanie

$$\phi^2 = \phi + 1$$

Tak więc proces ten wymaga liczby kroków, która rośnie wykładniczo ze względu na dane wejściowe. Z kolei potrzebna pamięć rośnie tylko liniowo ze względu na dane wejściowe, gdyż w każdej chwili przeprowadzanego obliczenia musimy śledzić tylko te węzły, które są powyżej nas w drzewie. Na ogół liczba kroków procesu rozgałęziającego się rekurencyjnie jest proporcjonalna do liczby węzłów w drzewie, a potrzebna pamięć jest proporcjonalna do maksymalnej wysokości drzewa.

Możemy też sformułować iteracyjny proces obliczający liczby Fibonacciego. Pomyśl polega na użyciu pary liczb całkowitych  $a$  i  $b$ , o wartościach początkowych  $a = \text{Fib}(1) = 1$  oraz  $b = \text{Fib}(0) = 0$ , i jednoczesnym powtarzaniu następujących przekształceń:

$$a \leftarrow a + b$$

$$b \leftarrow a$$

Można łatwo pokazać, że po  $n$ -krotnym zastosowaniu tych przekształceń  $a$  i  $b$  będą równe odpowiednio  $\text{Fib}(n + 1)$  i  $\text{Fib}(n)$ . Możemy więc obliczać liczby Fibonacciego iteracyjnie za pomocą takiej oto procedury:

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

Ta druga metoda obliczania  $\text{Fib}(n)$  jest iteracją liniową. Różnica w liczbie kroków wykonywanych w tych dwóch metodach — rosnącej tak szybko jak samo  $\text{Fib}(n)$  w przypadku pierwszej metody i liniowej ze względu na  $n$  w przypadku drugiej — jest ogromna, nawet dla niewielkich danych.

Nie należy jednak stąd wyciągać wniosku, że procesy rozgałęzające się rekurencyjnie są bezużyteczne. Gdy rozważymy procesy operujące na danych o strukturze hierarchicznej, zamiast na liczbach, wówczas okaże się, że rozgałęzająca się rekursja jest naturalnym i potężnym narzędziem<sup>32</sup>. Ale nawet w przypadku programów operujących na liczbach rozgałęzająca się rekursja może nam pomóc w zrozumieniu i konstruowaniu programów. Przykładowo, chociaż pierwsza procedura `fib` jest dużo mniej efektywna niż druga, jest ona jednak dużo prostsza, będąc niewiele więcej niż tłumaczeniem na Lisp definicji liczb Fibonacciego. Sformułowanie algorytmu iteracyjnego wymaga już zauważenia, że obliczenie można sprowadzić do iteracji z trzema zmiennymi stanu.

### Przykład: wydawanie reszty

W przeciwnieństwie do opisanego poniżej problemu dojście do iteracyjnego algorytmu obliczania liczb Fibonacciego wymaga tylko odrobiny sprytu. Rozważmy następujący problem: na ile różnych sposobów możemy wydać 1 dolara reszty, mając monety 50-, 25-, 10-, 5- i 1-centowe? Ogólnie, czy możemy napisać procedurę obliczającą, na ile sposobów można wydać daną kwotę reszty?

Problem ten ma proste rozwiązanie w postaci procedury rekurencyjnej. Przypuśćmy, że rodzaje dostępnych monet mamy jakoś uszeregowane. Wówczas zachodzi następująca zależność:

Liczba możliwych sposobów wydania reszty  $a$  przy wykorzystaniu  $n$  rodzajów monet to:

<sup>32</sup> Napomknęliśmy o tym w punkcie 1.1.3: sam interpreter używa rozgałęzającej się rekursji, obliczając wartości wyrażeń.

- liczba sposobów wydania reszty  $a$  przy użyciu wszystkich rodzajów monet z wyjątkiem pierwszego z nich plus
- liczba sposobów wydania reszty  $a - d$  przy użyciu wszystkich rodzajów monet, gdzie  $d$  jest nominałem monet pierwszego rodzaju.

Aby zrozumieć, dlaczego tak jest, należy zauważyć, że sposoby wydawania reszty mogą być podzielone na dwie grupy: te, w których nie użyto monet pierwszego rodzaju, i te, w których ich użyto. Dlatego też łączna liczba sposobów wydania reszty to liczba sposobów, na jakie możemy wydać resztę, nie używając monet pierwszego rodzaju, plus liczba sposobów, na jakie możemy wydać resztę, używając monet pierwszego rodzaju. Jednak druga z tych liczb jest równa liczbie sposobów, na jakie możemy wydać kwotę pozostałą po wydaniu jednej monety pierwszego rodzaju.

Tak więc problem wydawania określonej kwoty reszty możemy rekurencyjnie sprowadzić (zredukować) do problemu wydawania mniejszych kwot, używając mniejszej liczby rodzajów monet. Przypatrzmy się tej regule sprowadzania uważnie i przekonajmy się, że możemy za jej pomocą opisać algorytm, jeśli wyszczególnimy następujące zdegenerowane przypadki<sup>33</sup>:

- jeśli  $a$  jest równe dokładnie 0, to powinniśmy przyjąć, że mamy jeden sposób wydania reszty;
- jeśli  $a$  jest mniejsze od 0, to powinniśmy przyjąć, że mamy zero sposobów wydania reszty;
- jeśli  $n$  jest równe 0 [i  $a$  jest różne od 0; przyp. tłum.], to powinniśmy przyjąć, że mamy zero sposobów wydania reszty.

Powyższy opis możemy z łatwością przetłumaczyć na procedurę rekurencyjną:

```
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                      (- kinds-of-coins 1))
                  (cc (- amount
                            (first-denomination kinds-of-coins))
                      kinds-of-coins))))
```

<sup>33</sup> Sprawdź na przykład dokładnie, jak reguła stosuje się do problemu wydania 10 centów reszty, gdy mamy do dyspozycji monety 1- i 5-centowe.

---

```
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

(Argumentem procedury `first-denomination` jest liczba dostępnych rodzajów monet, a jej wynikiem jest nominal monet pierwszego rodzaju. W tym przypadku rodzaje monet są uszeregowane od największych do najmniejszych nominaliów, ale równie dobrze mogłyby to być dowolna inną kolejność). Możemy teraz odpowiedzieć na nasze początkowe pytanie o liczbę sposobów, na jakie można wydać dolara reszty:

```
(count-change 100)
292
```

Procedura `count-change` generuje proces rozgałęziający się rekurencyjnie i wykonujący nadmiarowe obliczenia, podobnie jak to było w przypadku naszej pierwszej implementacji procedury `fib`. (Uzyskanie wyniku 292 zajmie dobrą chwilę). Jednakże nie jest oczywiste, jak można by skonstruować lepszy algorytm obliczający wynik — pozostawiamy ten problem jako wyzwanie dla czytelnika. Fakt, iż procesy rozgałęziające się rekurencyjnie mogą być wysoce nieefektywne, choć często łatwe do opisania i zrozumienia, nasunął ludziom myśl, iż najlepsze wyniki można by uzyskać przez połączenie obu światów, konstruując „sprytne kompilatory”, które potrafiłyby przekształcać rozgałęziające się rekurencyjnie procedury w bardziej efektywne procedury obliczające takie same wyniki<sup>34</sup>.

### Ćwiczenie 1.11

Funkcja  $f$  jest zdefiniowana następująco:  $f(n) = n$  dla  $n < 3$  i  $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$  dla  $n \geq 3$ . Napisz procedurę obliczającą  $f$  za pomocą procesu rekurencyjnego. Napisz procedurę obliczającą  $f$  za pomocą procesu iteracyjnego.

---

<sup>34</sup> Jednym ze sposobów radzenia sobie z nadmiarowymi obliczeniami jest takie zorganizowanie obliczeń, aby w ich trakcie automatycznie tworzyć i wypełniać tablicę obliczanych wartości. Za każdym razem, gdy chcemy zastosować procedurę do argumentów, najpierw sprawdzamy w tablicy, czy wynik nie został już obliczony, a jeśli tak, to unikamy wykonywania nadmiarowych obliczeń. Strategia ta, znana jako *tablicowanie* (ang. *tabulation*) lub *spamiętywanie* (ang. *memoization*), jest prosta do zimplementowania. Za pomocą spamiętywania można czasem przekształcić procesy wymagające wykładniczej liczby kroków (takie jak `count-change`) w procesy wymagające czasu i pamięci rosnących liniowo ze względu na dane wejściowe. Zobacz ćwiczenie 3.27.

**Ćwiczenie 1.12**

Następujący układ liczb jest nazywany *trójkątem Pascala*:

		1		
	1	1		
1	2	1		
1	3	3	1	
1	4	6	4	1
...				

Wszystkie liczby na brzegach trójkąta są równe 1, a każda liczba w środku trójkąta jest sumą dwóch liczb znajdujących się bezpośrednio nad nią<sup>35</sup>. Napisz procedurę obliczającą elementy trójkąta Pascala za pomocą procesu rekurencyjnego.

**Ćwiczenie 1.13**

Udowodnij, że  $\text{Fib}(n)$  jest liczbą całkowitą najbliższą  $\phi^n / \sqrt{5}$ , gdzie  $\phi = (1 + \sqrt{5})/2$ .

Wskazówka: Niech  $\psi = (1 - \sqrt{5})/2$ . Korzystając z definicji liczb Fibonacciego (zob. punkt 1.2.2), udowodnij przez indukcję, że  $\text{Fib}(n) = (\phi^n - \psi^n) / \sqrt{5}$ .

**1.2.3. Rzędy wielkości**

Poprzednie przykłady pokazały, że procesy mogą się od siebie w znaczącym stopniu różnić pod względem ilości zużywanych zasobów obliczeniowych. Różnice te możemy w dogodny sposób uchwycić, opisując zgrubnie za pomocą *rzędów wielkości*, jak zmienia się ilość zasobów wymaganych przez proces wraz ze wzrostem wielkości danych wejściowych.

Niech  $n$  będzie parametrem określającym rozmiar problemu, a  $R(n)$  niech będzie maksymalną ilością zasobów wymaganych przez proces dla problemu rozmiaru  $n$ . W poprzednich przykładach  $n$  było liczbą, dla której obliczaliśmy wartość danej funkcji, ale są jeszcze inne możliwości. Jeżeli na przykład naszym celem jest obliczenie przybliżonej wartości pierwiastka kwadratowego z danej liczby, to  $n$  może określać, z dokładnością do ilu miejsc dziesiętnych należy obliczyć wynik. W przypadku mnożenia macierzy  $n$  może być liczbą wierszy macierzy. Na ogół może być wiele własności problemu, w odniesieniu do których może być wskazane przeanalizowanie danego procesu. Podobnie,  $R(n)$  może być miarą liczby użytych wewnętrznych rejestrów pamięci, miarą

<sup>35</sup> Elementy trójkąta Pascala są nazywane *współczynnikami dwumianowymi* (ang. *binomial coefficients*), ponieważ  $n$ -ty wiersz składa się ze współczynników występujących w rozwinięciu dwumianu  $(x + y)^n$ . Taki sposób obliczania współczynników dwumianu został przedstawiony w 1653 r. w inspirującej pracy Blaise'a Pascala *Traité du triangle arithmétique* poświęconej rachunkowi prawdopodobieństwa. Jak podaje Knuth [58], ten sam układ liczb pojawia się w pracy *Szu-yuen Yü-chien* („Nefrytowe zwierciadło czterech pierwiastków”) opublikowanej przez chińskiego matematyka Chu Shih-chieh w 1303 r., w pracach XII-wiecznego perskiego poety i matematyka Omara Khayyama oraz w pracach XII-wiecznego hinduskiego matematyka Bhāskara Áchārya.

liczby wykonanych elementarnych operacji maszynowych itp. W przypadku komputerów, które wykonują tylko ustaloną liczbę operacji na raz, wymagany czas będzie proporcjonalny do liczby wykonanych elementarnych operacji maszynowych.

Mówimy, że  $R(n)$  jest rzędu  $\Theta(f(n))$ , co zapisujemy jako  $R(n) = \Theta(f(n))$  („theta od  $f(n)$ ”), jeśli istnieją dodatnie stałe  $k_1$  i  $k_2$ , niezależne od  $n$ , takie że dla dostatecznie dużych wartości  $n$  zachodzi

$$k_1 f(n) \leq R(n) \leq k_2 f(n)$$

(Inaczej mówiąc, dla dużych  $n$  wartość  $R(n)$  znajduje się między  $k_1 f(n)$  i  $k_2 f(n)$ ).

Na przykład liczba kroków liniowego procesu rekurencyjnego obliczającego silnię (opisanego w punkcie 1.2.1) jest proporcjonalna do argumentu  $n$ . Tak więc liczba kroków wykonywanych przez proces jest rzędu  $\Theta(n)$ . Jak to już stwierdziliśmy, wymagana pamięć również jest rzędu  $\Theta(n)$ . W przypadku silni obliczanej iteracyjnie liczba kroków jest też rzędu  $\Theta(n)$ , ale wymagana pamięć jest rzędu  $\Theta(1)$  — tzn. jest stała<sup>36</sup>. Rozgałęzające się rekurencyjnie obliczenie liczb Fibonacciego wymaga  $\Theta(\phi^n)$  kroków i  $\Theta(n)$  pamięci, gdzie  $\phi$  jest złotym podziałem opisanym w punkcie 1.2.2.

Rzędy wielkości dają jedynie zgrubny opis zachowania procesu. Na przykład proces wymagający  $n^2$  kroków i proces wymagający  $1000n^2$  kroków, a także proces wymagający  $3n^2 + 10n + 17$  kroków — wszystkie wymagają liczby kroków rzędu  $\Theta(n^2)$ . Rząd wielkości jest jednakże dobrym wskazaniem co do tego, jakich zmian możemy się spodziewać w zachowaniu procesu, gdy będziemy zmieniać rozmiar problemu. W przypadku procesu opisanego jako rzędu  $\Theta(n)$  (czyli procesu liniowego) podwojenie rozmiaru problemu spowoduje, mniej więcej, podwojenie ilości zużytych zasobów. W przypadku procesu wykładowicznego każde zwiększenie rozmiaru problemu o stałą wielkość spowoduje pomnożenie ilości potrzebnych zasobów przez stały czynnik. W pozostałej części podrozdziału 1.2 zbadamy dwa algorytmy, których rząd wielkości jest logarytmiczny, a więc takie, w których dwukrotne zwiększenie rozmiaru problemu zwiększa ilość wymaganych zasobów o stałą ilość.

### Ćwiczenie 1.14

Narysuj drzewo ilustrujące proces obliczeniowy generowany przez procedurę `count-change` z punktu 1.2.2, gdy mamy 11 centów reszty do wydania. Jakiego rzędu są pamięć i liczba kroków wykonywanych przez ten proces w zależności od wielkości wydawanej reszty?

<sup>36</sup> Stwierdzenia te kryją w sobie duże uproszczenia. Jeśli na przykład traktujemy kroki procesu jako „operacje maszynowe”, to zakładamy, że liczba operacji maszynowych potrzebnych do wykonania, powiedzmy, mnożenia nie zależy od wielkości mnożonych liczb, co dla dostatecznie dużych liczb nie jest prawdą. Podobna uwaga tyczy się pamięci. Analiza procesów, tak samo jak ich konstrukcja i opisywanie, może być dokonana na różnych poziomach abstrakcji.

### Ćwiczenie 1.15

Sinus kąta (podanego w radianach) może być obliczany dla dostatecznie małych  $x$  za pomocą przybliżenia  $\sin x \approx x$ , a argument funkcji sinus może być zmniejszany za pomocą następującej tożsamości trygonometrycznej:

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

(Na potrzeby tego ćwiczenia przyjmujemy, że kąt jest „dostatecznie mały”, jeśli nie jest większy niż 0,1 radiana). Pomyśl ten jest zawarty w następujących procedurach:

```
(define (cube x) (* x x x))

(define (p x) (- (* 3 x) (* 4 (cube x)))))

(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0))))))
```

- (a) Ile razy w trakcie obliczania `(sine 12.15)` jest stosowana procedura `p`?  
 (b) Jaki jest rząd wielkości potrzebnej pamięci i liczby kroków procesu generowanego przez procedurę `sine` przy obliczaniu `(sine a)` (jako funkcja `a`)?

#### 1.2.4. Potęgowanie

Rozważmy problem obliczania potęgi danej liczby. Potrzebujemy procedury o dwóch argumentach: podstawie  $b$  i dodatnim całkowitym wykładniku  $n$ , której wynikiem jest  $b^n$ . Można to zrobić, stosując na przykład taką definicję rekurencyjną:

$$\begin{aligned} b^n &= b \cdot b^{n-1} \\ b^0 &= 1 \end{aligned}$$

która tłumaczy się natychmiast na procedurę

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1))))))
```

Jest to liniowy proces rekurencyjny, wymagający  $\Theta(n)$  kroków i  $\Theta(n)$  pamięci. Tak jak w przypadku silni, możemy od razu sformułować równoważną iterację liniową:

```
(define (expt b n)
  (expt-iter b n 1))

(define (expt-iter b counter product)
  (if (= counter 0)
```

```

product
(expt-iter b
            (- counter 1)
            (* b product)))

```

Ta wersja wymaga  $\Theta(n)$  kroków i  $\Theta(1)$  pamięci.

Możemy jednak obliczyć zadaną potęgę w mniejszej liczbie kroków, stosując wielokrotne podnoszenie do kwadratu. Zamiast obliczać na przykład  $b^8$  jako

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

możemy to zrobić za pomocą trzech mnożeń:

$$b^2 = b \cdot b$$

$$b^4 = b^2 \cdot b^2$$

$$b^8 = b^4 \cdot b^4$$

Ta metoda działa dobrze dla wykładników będących potęgami 2. Wielokrotne podnoszenie do kwadratu możemy też zastosować przy obliczaniu dowolnej potęgi, korzystając z następujących tożsamości:

$$b^n = (b^{n/2})^2 \quad \text{dla } n \text{ parzystych}$$

$$b^n = b \cdot b^{n-1} \quad \text{dla } n \text{ nieparzystych}$$

Metodę tę możemy zapisać w postaci takiej oto procedury:

```

(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))

```

gdzie `even?` jest predykatem sprawdzającym za pomocą procedury pierwotnej `remainder`, której wynikiem jest reszta z dzielenia, czy dana liczba całkowita jest parzysta:

```

(define (even? n)
  (= (remainder n 2) 0))

```

Proces rozwijający się zgodnie z `fast-expt` rośnie logarytmicznie ze względu na  $n$  zarówno w odniesieniu do pamięci, jak i liczby kroków. Aby się o tym przekonać, zauważmy, że obliczenie  $b^{2n}$  za pomocą `fast-expt` wymaga tylko jednego mnożenia więcej niż obliczenie  $b^n$ . Z każdym dodatkowym mnożeniem wykładnik potęgi, którą możemy obliczyć, podwaja się (w przybliżeniu). Tak więc liczba mnożeń, które należy wykonać, rośnie w zależności od wy-

kładnika  $n$  mniej więcej tak szybko jak logarytm  $n$  przy podstawie 2. Proces ten jest rzędu  $\Theta(\log n)$ <sup>37</sup>.

Różnica między rzędem  $\Theta(\log n)$  i  $\Theta(n)$  staje się uderzająca dla dużych  $n$ . Na przykład dla  $n = 1000$  procedura `fast-expt` wymaga 14 mnożeń<sup>38</sup>. Korzystając z pomysłu wielokrotnego podnoszenia do kwadratu, można również skonstruować iteracyjny algorytm podnoszenia do potęgi, wymagający logarytmicznej liczby kroków (zob. ćwiczenie 1.16), chociaż, jak to często jest w przypadku algorytmów iteracyjnych, nie zapisuje się go tak prosto jak algorytm rekurencyjny<sup>39</sup>.

### Ćwiczenie 1.16

Skonstruuj procedurę potęgowania tworzącą proces iteracyjny, który wykorzystuje wielokrotne podnoszenie do kwadratu i wykonuje, tak samo jak `fast-expt`, logarytmiczną liczbę kroków. (Wskazówka: Korzystając z tego, że  $(b^{n/2})^2 = (b^2)^{n/2}$ , zapamiętuj — oprócz wykładnika  $n$  i podstawy  $b$  — dodatkową zmienną stanu  $a$ , zaś zmianę stanu zdefiniuj w taki sposób, żeby iloczyn  $ab^n$  nie zmieniał się przy przejściu od stanu do stanu. Zakładamy, że na początku procesu  $a$  jest równe 1, a wynik jest równy wartości  $a$  uzyskanej na końcu procesu. Ogólnie mówiąc, umiejętność definiowania *wielkości niezmiennej* (ang. *invariant quantity*), która pozostaje niezmieniona przy zmianie stanu, jest potężną techniką konstruowania algorytmów iteracyjnych).

### Ćwiczenie 1.17

Algorytmy potęgujące przedstawione w tym punkcie opierają się na potęgowaniu za pomocą wielokrotnego mnożenia. W podobny sposób można mnożyć liczby całkowite za pomocą wielokrotnego dodawania. Następująca procedura mnożenia odpowiada procedurze `expt` (przy założeniu, że nasz język programowania umożliwia tylko dodawanie, a nie mnożenie):

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1))))))
```

Liczba wykonywanych przez ten algorytm kroków jest liniowa ze względu na  $b$ . Założymy teraz, że oprócz dodawania mamy dostępne operacje: `double`, podwajającą liczbę całkowitą, oraz `halve`, dzielącą (parzystą) liczbę całkowitą przez 2. Korzy-

<sup>37</sup> Dokładniej, wymagana liczba mnożeń jest o jeden mniejsza niż logarytm  $n$  przy podstawie 2 plus liczba jedynek w zapisie binarnym  $n$ . Liczba ta jest zawsze mniejsza niż dwa razy logarytm  $n$  przy podstawie 2. Z dowolnością wyboru wartości stałych  $k_1$  i  $k_2$  w definicji rzędu wielkości wynika, że dla procesu logarytmicznego podstawa logarytmu nie ma znaczenia, więc wszystkie takie procesy są opisywane jako rzędu  $\Theta(\log n)$ .

<sup>38</sup> Być może zastanawiasz się, dlaczego ktokolwiek miałby podnosić liczbę do 1000 potęgi. Zajrzyj do punktu 1.2.6.

<sup>39</sup> Wspomniany algorytm iteracyjny pochodzi ze starożytności. Pojawia się w *Chandah-sutra*, dziele napisanym przez Áchárya Pingala jeszcze przed 200 r. p.n.e. Pełne omówienie i analizę tej i innych metod potęgowania można znaleźć w pracy Knutha [59], punkt 4.6.3.

stając z tych operacji, skonstruuj procedurę mnożenia analogiczną do `fast-expt`, wykonującą logarytmiczną liczbę kroków.

### Ćwiczenie 1.18

Korzystając z wyników ćwiczeń 1.16 i 1.17, opracuj procedurę generującą iteracyjny proces mnożący dwie liczby całkowite za pomocą dodawania, podwajania wartości i dzielenia przez dwa, wykonującą logarytmiczną liczbę kroków<sup>40</sup>.

### Ćwiczenie 1.19

Istnieje sprytny algorytm obliczania liczb Fibonacciego w logarytmicznej liczbie kroków. Przypomnijmy sobie przekształcenie zmiennych stanu  $a$  i  $b$  w procesie `fib-iter` opisany w punkcie 1.2.2:  $a \leftarrow a + b$  i  $b \leftarrow a$ . Oznaczmy to przekształcenie przez  $T$ . Zauważmy, że  $n$ -krotne zastosowanie  $T$  do liczb 1 i 0 daje w wyniku parę  $\text{Fib}(n+1)$  i  $\text{Fib}(n)$ . Innymi słowy, liczby Fibonacciego są wynikiem zastosowania  $T^n$ , czyli  $n$ -tej potęgi przekształcenia  $T$ , do pary  $(1, 0)$ . Rozważmy teraz  $T$  jako szczególny przypadek, dla  $p = 0$  i  $q = 1$ , przekształcenia postaci  $T_{pq}$ , gdzie  $T_{pq}$  przekształca parę  $(a, b)$  zgodnie ze wzorami:  $a \leftarrow bq + aq + ap$  i  $b \leftarrow bp + aq$ . Pokaż, że efekt dwukrotnego zastosowania przekształcenia postaci  $T_{pq}$  jest taki sam jak pojedynczego zastosowania przekształcenia takiej samej postaci  $T_{p'q'}$ , i podaj, jakie są wartości  $p'$  i  $q'$  w zależności od  $p$  i  $q$ . Daje nam to bezpośredni sposób na podnoszenie takich przekształceń do kwadratu, a zatem możemy obliczyć  $T^n$  za pomocą wielokrotnego podnoszenia do kwadratu, tak jak w procedurze `fast-expt`. Połącz to wszystko i uzupełnij następującą procedurę, działającą w logarytmicznej liczbie kroków<sup>41</sup>:

```
(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   (??) ; oblicz p'
                   (??) ; oblicz q'
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1))))))
```

<sup>40</sup> Algorytm ten jest czasami nazywany „algorytmem rosyjskich chłopów” i był znany już w starożytności. Przykłady jego zastosowania można znaleźć w jednej z dwóch najstarszych istniejących prac matematycznych — papirusie Rhinda (będącym kopią jeszcze starszego dokumentu) spisany przez egipskiego skrybę imieniem A'h-mose ok. 1700 r. p.n.e.

<sup>41</sup> Ćwiczenie to zasugerował nam Joe Stoy, opierając się na przykładzie z książki Anne Kaldehaij [57].

### 1.2.5. Największe wspólne dzielniki

Największy wspólny dzielnik (NWD lub od ang. *greatest common divisor*) dwóch liczb całkowitych  $a$  i  $b$  jest zdefiniowany jako największa liczba całkowita, przez którą  $a$  i  $b$  dzielą się bez reszty. Przykładowo, NWD liczb 16 i 28 wynosi 4. W rozdziale 2, gdzie badamy, jak zaimplementować arytmetykę liczb wymiernych, będziemy musieli umieć obliczać NWD, aby skracać ułamki wymierne. (Aby skrócić ułamek wymierny musimy podzielić zarówno jego licznik, jak i mianownik przez ich NWD. Na przykład  $16/28$  skraca się do  $4/7$ ). Jeden ze sposobów na znalezienie NWD dwóch liczb całkowitych polega na rozłożeniu ich na czynniki pierwsze i wyszukaniu wspólnych czynników. Znany jest jednak słynny algorytm, który jest bardziej efektywny.

Pomysł algorytmu opiera się na obserwacji, że jeśli  $r$  jest resztą z dzielenia  $a$  przez  $b$ , to wspólne dzielniki  $a$  i  $b$  są dokładnie takie same jak wspólne dzielniki  $b$  i  $r$ . Tak więc możemy skorzystać ze wzoru

$$\text{NWD}(a, b) = \text{NWD}(b, r)$$

i sprowadzić problem obliczenia NWD do problemu obliczenia NWD kolejnych coraz mniejszych par liczb całkowitych. Na przykład:

$$\begin{aligned} \text{NWD}(206, 40) &= \text{NWD}(40, 6) \\ &= \text{NWD}(6, 4) \\ &= \text{NWD}(4, 2) \\ &= \text{NWD}(2, 0) \\ &= 2 \end{aligned}$$

NWD(206, 40) sprowadza się do NWD(2, 0), co jest równe 2. Można pokazać, że zaczynając od pary dowolnych dwóch dodatnich liczb całkowitych i powtarzając powyższe sprowadzenie, zawsze w końcu otrzymamy parę, której drugim elementem jest 0 — wówczas pierwszy jej element jest szukanym NWD. Taka metoda obliczania NWD jest znana jako *algorytm Euklidesa*<sup>42</sup>.

Algorytm Euklidesa można łatwo przedstawić w postaci procedury:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

<sup>42</sup> Algorytm Euklidesa nazywa się tak, ponieważ został opisany w *Elementach Euklidesa* (Księga 7, ok. 300 r. p.n.e.). Jak podaje Knuth [58], może być on uważany za najstarszy znany nietrywialny algorytm. Staroegipska metoda mnożenia (zob. ćwiczenie 1.18) jest z pewnością starsza, ale algorytm Euklidesa, jak wyjaśnia Knuth, jest najstarszym algorytmem, o którym wiadomo, że został przedstawiony w postaci ogólnej, a nie jako zestaw objaśniających przykładów.

Procedura ta generuje proces iteracyjny, w którym liczba kroków rośnie jak logarytm liczb, których dotyczy obliczenie.

Fakt, że algorytm Euklidesa wymaga logarytmicznej liczby kroków, wiąże się w ciekawy sposób z liczbami Fibonacciego:

**Twierdzenie Lamégo:** Jeśli algorytm Euklidesa wymaga  $k$  kroków, aby obliczyć NWD pary liczb, to mniejsza z liczb należących do tej pary jest większa lub równa  $k$ -tej liczbie Fibonacciego<sup>43</sup>.

Możemy użyć tego twierdzenia do otrzymania rzędu wielkości oszacowania algorytmu Euklidesa. Niech  $n$  będzie mniejszym z dwóch argumentów procedury. Jeśli proces wykonuje  $k$  kroków, to musi zachodzić  $n \geq \text{Fib}(k) \approx \phi^k / \sqrt{5}$ . Zatem liczba kroków algorytmu  $k$  rośnie jak logarytm  $n$  (przy podstawie  $\phi$ ). Stąd rzad wielkości to  $\Theta(\log n)$ .

### Ćwiczenie 1.20

Proces generowany przez procedurę zależy, oczywiście, od reguł stosowanych przez interpreter. Rozważmy na przykład podaną powyżej iteracyjną procedurę `gcd`. Założymy, że mamy ją zinterpretować, stosując normalną kolejność obliczania, omówioną w punkcie 1.1.5. (Reguła opisująca normalną kolejność obliczania `if` jest podana w ćwiczeniu 1.5). Wykorzystując model podstawieniowy (przy normalnej kolejności obliczania), zilustruj proces generowany przy obliczaniu (`gcd 206 40`) i wskaż, które operacje `remainder` są faktycznie wykonywane. Ile operacji `remainder` jest faktycznie wykonywanych przy normalnej, a ile przy stosowanej kolejności obliczania (`gcd 206 40`)?

#### 1.2.6. Przykład: testowanie pierwszości

W tym punkcie opisujemy dwa algorytmy sprawdzania, czy dana liczba całkowita  $n$  jest liczbą pierwszą — jeden rzędu  $\Theta(\sqrt{n})$ , a drugi algorytm „probabilistyczny”.

<sup>43</sup> Twierdzenie to zostało udowodnione w 1845 r. przez francuskiego matematyka i inżyniera Gabriela Lamégo znanego głównie ze swojego wkładu w fizykę matematyczną. Aby udowodnić to twierdzenie, rozważmy pary  $(a_k, b_k)$ , gdzie  $a_k \geq b_k$ , dla których algorytm Euklidesa kończy obliczenia w  $k$  krokach. Dowód opiera się na stwierdzeniu, że jeśli  $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$  są trzema kolejnymi parami pojawiającymi się w kolejnych krokach algorytmu, to musi zachodzić  $b_{k+1} \geq b_k + b_{k-1}$ . Aby sprawdzić to stwierdzenie, zauważmy, że pojedynczy krok algorytmu jest zdefiniowany jako zastosowanie następującego przekształcenia:  $a_{k-1} = b_k$ ,  $b_{k-1} = \text{reszta z dzielenia } a_k \text{ przez } b_k$ . Drugie równanie oznacza, że dla pewnego dodatniego całkowitego  $q$  mamy  $a_k = qb_k + b_{k-1}$ . Ponieważ  $q$  musi być równe co najmniej 1, więc  $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$ . Jednakże w poprzednim kroku algorytmu mieliśmy  $b_{k+1} = a_k$ . Stąd  $b_{k+1} = a_k \geq b_k + b_{k-1}$ , co dowodzi naszego stwierdzenia. Teraz możemy udowodnić twierdzenie przez indukcję względem  $k$  — czyli liczby kroków potrzebnych algorytmowi do zakończenia obliczeń. Dla  $k = 1$  twierdzenie jest prawdziwe, gdyż wymagamy jedynie, aby  $b$  było nie mniejsze niż  $\text{Fib}(1) = 1$ . Założymy teraz, że twierdzenie jest prawdziwe dla wszystkich liczb całkowitych mniejszych lub równych  $k$ . Pokażemy, że jest ono prawdziwe dla  $k + 1$ . Niech  $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$  będą parami liczb występującymi w kolejnych krokach algorytmu. Z naszego założenia indukcyjnego wynika, że  $b_{k-1} \geq \text{Fib}(k - 1)$  i  $b_k \geq \text{Fib}(k)$ . Tak więc stosując stwierdzenie, które przed chwilą udowodniliśmy, oraz definicję liczb Fibonacciego, otrzymujemy  $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k - 1) = \text{Fib}(k + 1)$ , co kończy dowód twierdzenia Lamégo.

styczny” rzędu  $\Theta(\log n)$ . Ćwiczenia na końcu tego punktu zawierają propozycje projektów programistycznych opartych na tych algorytmach.

### Poszukiwanie dzielników

Od czasów starożytnych matematyków fascynowały problemy dotyczące liczb pierwszych i wielu ludzi pracowało nad określeniem sposobów sprawdzania, czy dana liczba jest pierwsza. Jednym ze sposobów na sprawdzenie, czy liczba jest pierwsza, jest znalezienie dzielników tej liczby. Poniższy program znajduje najmniejszy całkowity dzielnik (większy niż 1) danej liczby  $n$ . Robi to w prosty sposób, sprawdzając podzielność  $n$  przez kolejne liczby całkowite, począwszy od 2.

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

Możemy sprawdzić, czy liczba jest pierwsza, w następujący sposób:  $n$  jest liczbą pierwszą wtedy i tylko wtedy, gdy jest swoim najmniejszym dzielnikiem.

```
(define (prime? n)
  (= n (smallest-divisor n)))
```

Pierwszy warunek w `find-divisor` opiera się na tym, że jeśli  $n$  nie jest liczbą pierwszą, to musi mieć dzielnik nie większy niż  $\sqrt{n}$ <sup>44</sup>. Oznacza to, że algorytm ten musi sprawdzić dzielniki tylko z zakresu od 1 do  $\sqrt{n}$ . W rezultacie liczba kroków wymaganych do stwierdzenia, czy  $n$  jest liczbą pierwszą, jest rzędu  $\Theta(\sqrt{n})$ .

### Test Fermata

Test pierwszości rzędu  $\Theta(\log n)$  opiera się na wyniku z teorii liczb, znanim jako małe twierdzenie Fermata<sup>45</sup>.

<sup>44</sup> Jeśli  $d$  jest dzielnikiem  $n$ , to  $n/d$  jest nim również. Jednak  $d$  i  $n/d$  nie mogą być jednocześnie większe niż  $\sqrt{n}$ .

<sup>45</sup> Pierre de Fermat (1601–1665) jest uważany za twórcę współczesnej teorii liczb. Uzyskał wiele ważnych wyników z teorii liczb, ale zwykle ogłaszał same wyniki bez dowodów. Małe twierdzenie Fermata zostało sformułowane w liście, który napisał w 1640 r. Pierwszy dowód został opublikowany przez Eulera w 1736 r. (identyczny wcześniejszy dowód odkryto wśród nieopublikowanych rękopisów Leibniza). Najsłynniejszy z wyników Fermata — znany jako

**Małe twierdzenie Fermata:** Jeśli  $n$  jest liczbą pierwszą, zaś  $a$  jest dowolną dodatnią liczbą całkowitą mniejszą od  $n$ , to  $a$  podniesione do potęgi  $n$  przystaje do  $a$  modulo  $n$ .

(Mówimy, że dwie liczby *przystają do siebie modulo  $n$* , jeśli obydwie mają taką samą resztę z dzielenia przez  $n$ . Reszta z dzielenia liczby  $a$  przez  $n$  jest także nazywana *resztą z  $a$  modulo  $n$*  lub krócej  *$a$  modulo  $n$* ).

Ogólnie mówiąc, jeśli  $n$  nie jest liczbą pierwszą, to większość liczb  $a < n$  nie będzie spełniać powyższej własności. To stwierdzenie prowadzi do następującego algorytmu testującego pierwszość: Mając daną liczbę  $n$ , wylosujmy liczbę losową  $a < n$  i obliczmy resztę z  $a^n$  modulo  $n$ . Jeśli jest ona różna od  $a$ , to  $n$  na pewno nie jest liczbą pierwszą. Jeśli jest ona równa  $a$ , to mamy duże szanse na to, że  $n$  jest liczbą pierwszą. Następnie wylosujmy inną liczbę losową  $a$  i sprawdźmy ją. Jeśli również spełnia ona podane równanie, to możemy być jeszcze bardziej pewni, że  $n$  jest liczbą pierwszą. Sprawdzając więcej liczb  $a$ , możemy zwiększyć naszą pewność co do poprawności wyniku. Algorytm ten jest znany jako test Fermata.

Aby zaimplementować test Fermata, potrzebujemy procedury, która podnosi jedną liczbę do potęgi modulo druga liczba (tzw. potęgowanie modularne):

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))
```

Przypomina ona bardzo procedurę *fast-expt* z punktu 1.2.4. Stosuje wielokrotne podnoszenie do kwadratu i dlatego wykonuje logarytmiczną, względem wykładownika, liczbę kroków<sup>46</sup>.

Test Fermata polega na wybraniu liczby losowej  $a$  z zakresu od 1 do  $n - 1$  włącznie i sprawdzeniu, czy reszta z  $a$  do  $n$ -tej potęgi modulo  $n$  jest równa  $a$ .

wielkie twierdzenie Fermata — został przez niego zanotowany w 1637 r. na marginesie jego egzemplarza książki (greckiego matematyka Diofantosa, żyjącego w III w.) *Arytmetyka* wraz z uwagą „Znalazłem naprawdę zadziwiający dowód, ale ten margines jest zbyt wąski, aby go zmieścić”. Znalezienie dowodu wielkiego twierdzenia Fermata stało się jednym z najbardziej znanych wyzwań w teorii liczb. Pełny dowód został w końcu podany w 1995 r. przez Andrew Wilesa z Princeton University.

<sup>46</sup> Kroki redukcyjne, gdy wykładownik  $e$  jest większy niż 1, opierają się na tym, że dla dowolnych liczb całkowitych  $x$ ,  $y$  i  $m$  możemy wyznaczyć resztę z  $x$  razy  $y$  modulo  $m$ , obliczając osobno reszty  $x$  modulo  $m$  i  $y$  modulo  $m$ , mnożąc je przez siebie i biorąc resztę z wyniku mnożenia modulo  $m$ . Jeśli na przykład  $e$  jest parzyste, obliczamy resztę z  $b^{e/2}$  modulo  $m$ , podnosimy ją do kwadratu i bierzemy resztę modulo  $m$ . Technika ta jest użyteczna, gdyż pozwala nam na wykonywanie obliczeń bez operowania liczbami dużo większymi od  $m$ . (Porównaj ćwiczenie 1.25).

Liczba losowa  $a$  jest wybierana za pomocą procedury `random`, co do której zakładamy, że jest w języku Scheme procedurą pierwotną. `Random` daje w wyniku nieujemną liczbę całkowitą, mniejszą od swojego argumentu, będącego liczbą całkowitą. Stąd, aby otrzymać liczbę losową z zakresu od 1 do  $n - 1$ , należy wywołać `random` z argumentem równym  $n - 1$  i dodać 1 do otrzymanego wyniku:

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

Następująca procedura wykonuje zadaną (jako argument) liczbę razy test Fermata. Jej wartością jest prawda, jeśli każdy z testów się powiodł, lub fałsz w przeciwnym razie.

```
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

### Metody probabilistyczne

Test Fermata różni się charakterem od najbardziej znanych algorytmów, w przypadku których mamy pewność, że obliczany wynik jest poprawny. Tutaj otrzymywany wynik jest tylko prawdopodobnie poprawny. Ścisłej mówiąc, jeśli  $n$  choć raz nie przejdzie testu Fermata, to możemy być pewni, że  $n$  nie jest liczbą pierwszą. Jednak fakt, że  $n$  przeszło test, choć jest niezwykle silną przesłanką, nie gwarantuje, że  $n$  jest liczbą pierwszą. Chcielibyśmy powiedzieć, że dla dowolnej liczby  $n$ , jeśli wykonamy test odpowiednią liczbę razy i  $n$  za każdym razem przejdzie test, to prawdopodobieństwo błędного wyniku w naszym teście pierwszości może być dowolnie małe.

Niestety, takie stwierdzenie nie jest całkiem prawdziwe. Istnieją liczby, które oszukują test Fermata: takie liczby  $n$ , które nie są liczbami pierwszymi, a mimo to mają tę własność, że  $a^n$  przystaje do  $a$  modulo  $n$  dla wszystkich liczb całkowitych  $a < n$ . Liczby takie są niezwykle rzadkie, a więc test Fermata jest w praktyce całkiem wiarygodny<sup>47</sup>. Są odmiany testu Fermata, których nie można oszukać. W testach tych, tak jak w metodzie Fermata, sprawdza się,

<sup>47</sup> Liczby, które oszukują test Fermata, są nazywane *liczbami Carmichaela* i niewiele o nich wiadomo oprócz tego, że są niezwykle rzadkie. Poniżej 100 000 000 jest 255 liczb Carmichaela. Kilka najmniejszych z nich to: 561, 1105, 1729, 2465, 2821 i 6601. Przy testowaniu pierwszości bardzo dużych losowo wybranych liczb szansa natknięcia się na liczbę oszukującą test Fermata jest mniejsza niż szansa na to, że promieniowanie kosmiczne spowoduje, iż komputer popełni błąd w trakcie wykonywania „poprawnego” algorytmu. Uważanie, że algorytm jest nieodpowiedni z pierwszego powodu, ale nie z drugiego ilustruje różnicę między matematyką i inżynierią.

czy liczba  $n$  jest liczbą pierwszą, wybierając losową liczbę całkowitą  $a < n$  i sprawdzając pewien warunek zależny od  $n$  i  $a$ . (Przykład takiego testu można znaleźć w ćwiczeniu 1.28). Z drugiej strony, w przeciwnieństwie do testu Fermata, można udowodnić dla dowolnego  $n$ , że jeśli  $n$  nie jest liczbą pierwszą, to warunek nie jest spełniony dla większości liczb całkowitych  $a < n$ . Zatem, jeśli  $n$  przejdzie test dla pewnej losowo wybranej liczby  $a$ , to mamy większe szanse na to, że  $n$  jest liczbą pierwszą. Jeśli  $n$  przejdzie test dla dwóch losowo wybranych wartości  $a$ , to szansa, że  $n$  jest liczbą pierwszą, jest większa niż  $3/4$ . Powtarzając test dla większej liczby losowo wybranych wartości  $a$ , możemy uzyskać dowolnie małe prawdopodobieństwo błędu.

Istnienie testów, dla których można udowodnić, że szansa na popełnienie pomyłki może być dowolnie mała, rozbudziło zainteresowanie algorytmami tego typu, znanyimi jako *algorytmy probabilistyczne* (ang. *probabilistic algorithms*). Wiele badań jest poświęconych tej tematyce, a algorytmy probabilistyczne znalazły owocne zastosowania w wielu dziedzinach<sup>48</sup>.

### Ćwiczenie 1.21

Użyj procedury `smallest-divisor` do znalezienia najmniejszych dzielników następujących liczb: 199, 1999, 19999.

### Ćwiczenie 1.22

Większość implementacji Lispu udostępnia procedurę pierwotną `runtime`, której wynikiem jest liczba całkowita określająca ilość czasu pracy systemu (mierzona np. w mikrosekundach). Następująca procedura `timed-prime-test`, wywoływana z argumentem  $n$  będącym liczbą całkowitą, wypisuje  $n$  i sprawdza, czy jest to liczba pierwsza. Jeśli  $n$  jest liczbą pierwszą, to procedura wypisuje trzy gwiazdki, a po nich ilość czasu zużytego na wykonanie testu.

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))
```

<sup>48</sup> Jedno z najbardziej frapujących zastosowań probabilistycznych testów pierwszości dotyczyło kryptografii. Mimo że obecnie rozłożenie 200-cyfrowej liczby na czynniki pierwsze jest niewykonalne, sprawdzenie za pomocą testu Fermata, czy jest to liczba pierwsza, zajmuje kilka sekund. Fakt ten jest podstawą techniki tworzenia „szyfrów nie do złamania”, zaproponowanej przez Rivesta, Shamira i Adlemana [88]. Opracowany przez nich *algorytm RSA* jest szeroko stosowany do podnoszenia bezpieczeństwa komunikacji elektronicznej. Ze względu na to i po krewne osiągnięcia badania nad liczbami pierwszymi — kiedyś uważane za typowy przykład działu „czystej” matematyki, badanej tylko dla samego badania — okazują się mieć teraz ważne praktyczne zastosowanie w kryptografii, elektronicznym przesyłaniu pieniędzy i wyszukiwaniu informacji.

---

```
(define (report-prime elapsed-time)
  (display "***")
  (display elapsed-time))
```

Korzystając z tej procedury, napisz procedurę `search-for-primes` sprawdzającą, czy kolejne liczby nieparzyste z zadanego zakresu są liczbami pierwszymi. Użyj swojej procedury do znalezienia trzech najmniejszych liczb pierwszych większych od: 1000; 10 000; 100 000; 1 000 000. Zauważ, jakie czasy są potrzebne do przetestowania każdej z liczb pierwszych. Ze względu na to, że algorytm testujący jest rzędu  $\Theta(\sqrt{n})$ , powinieneś się spodziewać, że przetestowanie liczb pierwszych równych około 10 000 powinno zająć  $\sqrt{10}$  razy więcej czasu niż przetestowanie liczb pierwszych równych około 1000. Czy Twoje pomiary czasu to potwierdzają? Jak dokładnie pomiary dla 100 000 i 1 000 000 potwierdzają przewidywany rzad  $\sqrt{n}$ ? Czy Twoje wyniki są zgodne z mniemaniem, że programy na Twoim komputerze działają w czasie proporcjonalnym do liczby kroków wykonywanych w trakcie obliczeń?

### Ćwiczenie 1.23

Procedura `smallest-divisor` przedstawiona na początku tego punktu wykonuje mnóstwo niepotrzebnych testów — po sprawdzeniu, czy liczba nie dzieli się przez 2, nie ma powodu sprawdzać, czy dzieli się przez jakąkolwiek większą liczbę parzystą. Tak więc argumentami `test-divisor` nie powinny być liczby 2, 3, 4, 5, 6, ..., ale raczej 2, 3, 5, 7, 9, .... Aby zaimplementować taką zmianę, zdefiniuj procedurę `next`, której wynikiem dla argumentu równego 2 jest 3, a w pozostałych przypadkach wartość argumentu plus 2. Zmodyfikuj procedurę `smallest-divisor` tak, aby wyoływała (`next test-divisor`) zamiast (`+ test-divisor 1`). Wykonaj testy dla wszystkich 12 liczb pierwszych wyznaczonych w ćwiczeniu 1.22, używając procedury `timed-prime-test` zawierającej zmienioną wersję `smallest-divisor`. Skoro ta modyfikacja zmniejsza liczbę wykonywanych testów o połowę, powinieneś spodziewać się, że program będzie działał dwa razy szybciej. Czy to oczekiwanie się sprawdziło? Jeśli nie, to jaki jest stosunek zaobserwowanych szybkości działania tych dwóch algorytmów i jak wyjaśnić fakt, że jest on różny od 2?

### Ćwiczenie 1.24

Zmodyfikuj procedurę `timed-prime-test` z ćwiczenia 1.22 tak, aby korzystała z `fast-prime?` (metody Fermata), i przetestuj każdą z 12 liczb pierwszych wyznaczonych w tym ćwiczeniu. Ponieważ test Fermata jest rzędu  $\Theta(\log n)$ , jakiej zatem spodziewałbyś się różnic w czasach testowania dla liczb pierwszych z okolic 1000 i 1 000 000? Czy wyniki to potwierdzają? Czy możesz wyjaśnić powstałe rozbieżności?

### Ćwiczenie 1.25

Liz P. Haker narzeka, że pisząc procedurę `expmod`, wykonaliśmy mnóstwo niepotrzebnej pracy. Skoro — mówi ona — wiemy już, jak obliczać potęgi, moglibyśmy po prostu napisać

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Czy ma rację? Czy taka procedura równie dobrze nadawałaby się do naszego szybkiego testowania liczb pierwszych? Odpowiedź uzasadnij.

### Ćwiczenie 1.26

Ludwik Myślicielak ma duże problemy ze zrobieniem ćwiczenia 1.24. Wydaje się, że jego procedura testująca `fast-prime?` działa wolniej niż `prime?`. Ludwik wzywa na pomoc swoją przyjaciółkę Ewę Lu Ator. W trakcie sprawdzania programu Ludwika odkrywają, iż zmienił on procedurę `expmod` tak, że wykonuje ona jawne mnożenie, zamiast wywołać `square`:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                      (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))
```

„Nie widzę różnicy” — mówi Ludwik. „A ja owszem” — mówi Ewa. „Pisząc taką procedurę, zmieniłeś proces rzędu  $\Theta(\log n)$  w proces rzędu  $\Theta(n)$ ”. Wyjaśnij tę wypowiedź.

### Ćwiczenie 1.27

Pokaż, że liczby Carmichaela wymienione w przypisie 47 rzeczywiście oszukują test Fermata. To znaczy, napisz procedurę, która dla danego argumentu  $n$  sprawdza, czy  $a^n$  przystaje do  $a$  modulo  $n$  dla każdego  $a < n$ ; następnie wykonaj tę procedurę dla podanych liczb Carmichaela.

### Ćwiczenie 1.28

Jeden z wariantów testu Fermata, którego nie można oszukać, jest nazywany *testem Millera-Rabina* [76, 83] [oraz [16], p. 33.8; przyp. tłum.]. Opiera się on na alternatywnej postaci małego twierdzenia Fermata, mówiącej, że jeżeli  $n$  jest liczbą pierwszą, zaś  $a$  jest dowolną dodatnią liczbą całkowitą mniejszą od  $n$ , to  $a$  podniesione do potęgi  $(n-1)$  przystaje modulo  $n$  do 1. Aby sprawdzić za pomocą testu Millera-Rabina, czy  $n$  jest liczbą pierwszą, wybieramy liczbę losową  $a < n$  i podnosimy  $a$  do  $(n-1)$ -szej potęgi modulo  $n$ , korzystając z procedury `expmod`. Jednakże za każdym razem, gdy w procedurze `expmod` podnosimy liczbę do kwadratu, sprawdzamy, czy nie znaleźliśmy „nietrywialnego pierwiastka kwadratowego z 1 modulo  $n$ ”, tzn. liczby różnej od 1 i  $n-1$ , której kwadrat przystaje do 1 modulo  $n$ . Można pokazać, że jeżeli taki nietrywialny pierwiastek z 1 istnieje, to  $n$  nie jest liczbą pierwszą. Można też pokazać, że jeżeli  $n$  jest liczbą nieparzystą, która nie jest liczbą pierwszą, to przy najmniej dla połowy liczb  $a < n$  obliczanie w ten sposób  $a^{n-1}$  ujawni nietrywialny pierwiastek kwadratowy z 1 modulo  $n$ . (Oto dlaczego testu Millera-Rabina nie można oszukać). Tak zmodyfikuj procedurę `expmod`, aby sygnalizowała znalezienie nietrywialnego pierwiastka kwadratowego z 1, a następnie użyj jej do zaimplementowania

testu Millera-Rabina za pomocą procedury analogicznej do `fermat-test`. Sprawdź swoją procedurę, testując różne znane liczby pierwsze i liczby nie będące liczbami pierwszymi. Wskazówka: Jednym z dogodnych sposobów na to, aby procedura `expmod` „dawała znak”, jest wynik równy 0.

### 1.3. Formułowanie abstrakcji za pomocą procedur wyższych rzędów

Widzieliśmy, że procedury są w rzeczywistości abstrakcjami, które opisują złożone operacje na liczbach w oderwaniu od konkretnych liczb. Kiedy na przykład piszemy

```
(define (cube x) (* x x x))
```

nie chodzi nam o sześćian jakiejś konkretnej liczby, ale o metodę obliczania sześcielanu dowolnej liczby. Oczywiście moglibyśmy poradzić sobie bez definiowania tej procedury, pisząc zawsze takie wyrażenia, jak:

```
(* 3 3 3)  
(* x x x)  
(* y y y)
```

i nie odwołując się nigdy wprost do `cube`. Byłoby to poważne działanie na naszą niekorzyść — zmuszające nas zawsze do pracy na poziomie poszczególnych pierwotnych operacji języka (w tym przypadku mnożenia) zamiast posługiwanego się operacjami wyższego rzędu. Nasze programy mogłyby obliczać sześciiany, ale w naszym języku brakowałoby pojęcia sześcielanu. Jedna z rzeczy, których powinniśmy wymagać od języka programowania o dużej sile wyrazu, to możliwość budowania abstrakcji poprzez nazywanie ogólnych wzorców, a następnie bezpośrednie odwoływanie się do nich. To właśnie zapewniają procedury. Dlatego też wszystkie języki programowania, oprócz najprymitywniejszych, zawierają mechanizmy definiowania procedur.

Jednak nawet w przypadku przetwarzania numerycznego będziemy mieli poważnie uszczuploną możliwość tworzenia abstrakcji, jeśli jesteśmy ograniczeni do procedur, których argumenty muszą być liczbami. Często ten sam wzorzec programowania będzie wykorzystywany w wielu różnych procedurach. Chcąc wyrazić takie wzorce jako pojęcia, będziemy musieli tworzyć procedury, których argumenty lub wyniki same są procedurami. Procedury operujące na procedurach są nazywane *procedurami wyższych rzędów* (ang. *higher-order procedures*). W tym podrozdziale pokażemy, jak procedury wyższych rzędów mogą służyć jako potężne mechanizmy abstrakcji, znacznie zwiększając siłę wyrazu naszego języka.

### 1.3.1. Procedury jako argumenty

Rozważmy następujące trzy procedury. Pierwsza procedura oblicza sumę liczb całkowitych od a do b:

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

Druga procedura oblicza sumę sześciianów liczb całkowitych z danego zakresu:

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

Trzecia procedura oblicza sumy częściowe szeregu

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots$$

zbieżnego (bardzo powoli) do  $\pi/8$ <sup>49</sup>:

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

Te trzy procedury powstały wyraźnie według tego samego wzorca. W większej swej części są identyczne, a różnią się jedynie nazwą procedury, funkcją (zależną od a) użytą do obliczania sumowanych elementów i funkcją określającą następną wartość a. Każda z tych procedur mogłaby powstać przez wypełnienie pustych miejsc w tym samym szablonie:

```
(define (<nazwa> a b)
  (if (> a b)
      0
      (+ (<wyrażenie> a)
          (<nazwa> (<następna wartość> a) b))))
```

Występowanie takiego ogólnego wzorca to niezbity dowód na to, że kryje się tu użyteczna abstrakcja, którą należy uchwycić. Faktycznie, matematycy

<sup>49</sup> Szereg ten, zwykle zapisywany w równoważnej postaci  $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ , pochodzi od Leibniza. W punkcie 3.5.3 zobaczymy, jak można go użyć w kilku wymyślnych sztuczkach numerycznych.

dawno już zidentyfikowali abstrakcję *sumowania szeregów* i do oznaczenia jej wprowadzili „notację sigma”; na przykład:

$$\sum_{n=a}^b f(n) = f(a) + \cdots + f(b)$$

Siła notacji sigma polega na tym, że pozwala matematykom skupić się na samym pojęciu sumowania, a nie na poszczególnych sumach — mogą oni na przykład formułować ogólnie stwierdzenia dotyczące sum, niezależne od tego, jakie szeregi są sumowane.

Jako konstruktory programów, podobnie chcielibyśmy, aby nasz język był na tyle silny, żeby można było w nim zapisać procedurę wyrażającą pojęcie samego sumowania, a nie tylko procedury obliczające poszczególne sumy. W naszym języku proceduralnym możemy zrobić to od razu, zastępując w powyższym szablonie „miejscą do wypełnienia” parametrami formalnymi:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
          (sum term (next a) next b))))
```

Zauważmy, że argumentami sum są dolne i górne ograniczenia a i b oraz procedury term i next. Z procedury sum możemy korzystać tak jak z każdej innej. Możemy na przykład jej użyć (razem z procedurą inc zwiększającą swój argument o 1) do zdefiniowania sum-cubes:

```
(define (inc n) (+ n 1))

(define (sum-cubes a b)
  (sum cube a inc b))
```

Z pomocą tej procedury możemy obliczyć sumę sześciianów liczb całkowitych od 1 do 10:

```
(sum-cubes 1 10)
3025
```

Procedurę sum-integers możemy zdefiniować, używając procedury identycznościowej do określenia sumowanych elementów:

```
(define (identity x) x)

(define (sum-integers a b)
  (sum identity a inc b))
```

Wówczas możemy zsumować liczby całkowite od 1 do 10 w następujący sposób:

```
(sum-integers 1 10)
55
```

W ten sam sposób możemy również zdefiniować pi-sum<sup>50</sup>:

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

Z pomocą tych procedur możemy obliczyć przybliżoną wartość  $\pi$ :

```
(* 8 (pi-sum 1 1000))
3.139592655589783
```

Teraz, gdy mamy już zdefiniowaną procedurę sum, możemy jej użyć jako elementu składowego przy formułowaniu dalszych pojęć. Na przykład całka oznaczona funkcji  $f$  na przedziale od  $a$  do  $b$  może być przybliżana w obliczeniach numerycznych za pomocą wzoru

$$\int_a^b f = \left[ f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

dla małych wartości  $dx$ . Wzór ten możemy zapisać bezpośrednio w postaci takiej oto procedury:

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))

(integral cube 0 1 0.01)
.24998750000000042

(integral cube 0 1 0.001)
.24999875000001
```

(Dokładna wartość całki funkcji cube na przedziale od 0 do 1 wynosi 1/4).

<sup>50</sup> Zauważmy, że definicje procedur pi-next i pi-term umieszczone, za pomocą struktury blokowej (zob. punkt 1.1.8), wewnętrz pi-sum, gdyż jest mało prawdopodobne, aby procedury te były przydatne do jakiegokolwiek innego celu. W punkcie 1.3.2 zobaczymy, jak można się ich całkiem pozbyć.

### Ćwiczenie 1.29

Metoda Simpsona jest dokładniejszą, od przedstawionej powyżej, metodą całkowania numerycznego. W metodzie Simpsona całka funkcji  $f$  na przedziale od  $a$  do  $b$  jest przybliżana wzorem

$$\frac{h}{3} [y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n]$$

gdzie  $h = (b-a)/n$ ,  $n$  jest nieparzystą liczbą całkowitą, a  $y_k = f(a+kh)$ . (Zwiększając  $n$ , zwiększamy dokładność przybliżenia). Zdefiniuj procedurę, której argumentami są  $f$ ,  $a$ ,  $b$  i  $n$ , wynikiem zaś jest całka obliczona metodą Simpsona. Użyj swojej procedury do całkowania `cube` na przedziale od 0 do 1 (dla  $n = 100$  i  $n = 1000$ ) i porównaj uzyskane wyniki z wynikami przedstawionej powyżej procedury `integral`.

### Ćwiczenie 1.30

Przedstawiona wcześniej procedura `sum` generuje liniowy proces rekurencyjny. Procedurę tę można przepisać tak, aby dodawanie było wykonywane iteracyjnie. Pokaż, jak to można zrobić, uzupełniając brakujące wyrażenia w następującej definicji:

```
(define (sum term a next b)
  (define (iter a result)
    (if (??)
        (??)
        (iter (??) (??))))
  (iter (??) (??)))
```

### Ćwiczenie 1.31

(a) Procedura `sum` jest tylko najprostszą z ogromnej liczby podobnych abstrakcji, jakie można ująć jako procedury wyższych rzędów<sup>51</sup>. Napisz analogiczną procedurę `product` obliczającą iloczyn wartości funkcji w punktach z zadanego zakresu. Pokaż, jak można za pomocą tej procedury zdefiniować procedurę `factorial` (obliczającą silnie). Zastosuj również procedurę `product` do obliczenia przybliżonej wartości  $\pi$  na podstawie wzoru<sup>52</sup>

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

(b) Jeśli Twоя procedura `product` generuje proces rekurencyjny, to napisz taką, która generuje proces iteracyjny. Jeśli zaś generuje proces iteracyjny, to napisz taką, która generuje proces rekurencyjny.

<sup>51</sup> Celem ćwiczeń 1.31–1.33 jest pokazanie siły wyrazu, jaką możemy osiągnąć, używając odpowiedniej abstrakcji obejmującej wiele pozornie zupełnie różnych operacji. Chociaż akumulacja i filtrowanie to eleganckie pojęcia, jednak używając ich tutaj, mamy trochę związanego ręce, gdyż nie znamy jeszcze struktur danych, które umożliwiłyby nam wygodne łączenie tych abstrakcji. Wróćmy do tych pojęć w punkcie 2.2.3, gdzie pokażemy, jak można używać *ciągów* jako interfejsów przy łączeniu filtrów i akumulatorów w celu budowania jeszcze potężniejszych abstrakcji. Zobaczmy wówczas, na co w rzeczywistości stać te metody, jako potężne i eleganckie podejścia do konstruowania programów.

<sup>52</sup> Wzór ten wymyślił XVII-wieczny angielski matematyk John Wallis.

### Ćwiczenie 1.32

(a) Pokaż, że procedury `sum` i `product` (ćwiczenie 1.31) są szczególnymi przypadkami jeszcze bardziej ogólnej procedury `accumulate`, która odzwierciedla pojęcie *akumulacji* polegające na połączeniu zestawu elementów za pomocą pewnej ogólnej funkcji `kumulującej`:

`(accumulate combiner null-value term a next b)`

Procedura ta ma te same argumenty co `sum` i `product`, określające elementy i zakres, argument w postaci procedury `combiner` (o dwóch argumentach) określający, jak bieżący element ma być dołączony do skumulowanej wartości poprzednich elementów, oraz argument `null-value` określający wartość początkową, której należy użyć, gdy elementy się skończą. Zapisz procedurę `accumulate` i pokaż, w jaki sposób zarówno `sum`, jak i `product` mogą być zdefiniowane jako proste wywołania `accumulate`.

(b) Jeśli Twoja procedura `accumulate` generuje proces rekurencyjny, to napisz taką, która generuje proces iteracyjny. Jeśli zaś generuje proces iteracyjny, to napisz taką, która generuje proces rekurencyjny.

### Ćwiczenie 1.33

Możesz uzyskać jeszcze ogólniejszą wersję procedury `accumulate` (ćwiczenie 1.32), wprowadzając pojęcie *filtrowanych* elementów. To znaczy, połącz jedynie te elementy, uzyskane z wartości należących do zakresu, które spełniają określony warunek. Powstała w ten sposób abstrakcja `filtered-accumulate` ma te same argumenty co `accumulate` oraz dodatkowo jednoargumentowy predykat określający filtr. Zapisz `filtered-accumulate` w postaci procedury. Pokaż, jak za pomocą `filtered-accumulate` można wyrazić następujące funkcje:

(a) sumę kwadratów liczb pierwszych należących do przedziału od  $a$  do  $b$  (zakładając, że masz już napisany predykat `prime?` sprawdzający, czy liczba jest pierwszą);

(b) iloczyn wszystkich dodatnich liczb całkowitych mniejszych od  $n$  i względnie pierwszych z  $n$  (tzn. wszystkich dodatnich liczb całkowitych  $i < n$ , takich że  $\text{NWD}(i, n) = 1$ ).

#### 1.3.2. Tworzenie procedur za pomocą lambda-abstrakcji

Wydaje się niezwykle dziwne, że korzystając z procedury `sum`, tak jak to robiliśmy w punkcie 1.3.1, musimy definiować tak trywialne procedury, jak `pi-term` i `pi-next`, tylko po to, żeby przekazać je jako argumenty do procedury wyższego rzędu. Zamiast definiować `pi-term` i `pi-next`, poręczniej byłoby móc bezpośrednio określić „procedurę, której wynikiem jest jej argument powiększony o 4” i „procedurę, której wynikiem jest odwrotność jej argumentu pomnożonego przez jego wartość plus 2”. Możemy to zrobić za pomocą formy specjalnej `lambda` tworzącej procedurę, zwanej *lambda-abstrakcją*. Używając `lambda-abstrakcji`, możemy zapisać to, co chcemy, jako

`(lambda (x) (+ x 4))`

oraz

```
(lambda (x) (/ 1.0 (* x (+ x 2))))
```

Wówczas naszą procedurę pi-sum możemy wyrazić, bez definiowania żadnych procedur pomocniczych, jako

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
    a
    (lambda (x) (+ x 4))
    b))
```

Ponownie, używając lambda-abstrakcji, możemy bez definiowania procedury pomocniczej add-dx zapisać procedurę integral:

```
(define (integral f a b dx)
  (* (sum f
    (+ a (/ dx 2.0))
    (lambda (x) (+ x dx))
    b)
  dx))
```

Ogólnie mówiąc, lambda jest używana tak jak define, z tym że tworzona procedura nie ma nazwy:

```
(lambda (<parametry formalne>) <treść>)
```

Powstała procedura jest w takim samym stopniu procedurą jak procedury two-rzone za pomocą define. Jedyna różnica polega na tym, że nie jest ona związana z żadną nazwą w środowisku. W rzeczywistości

```
(define (plus4 x) (+ x 4))
```

jest równoważne

```
(define plus4 (lambda (x) (+ x 4)))
```

Wyrażenia z lambda-abstrakcją (lambda-wyrażenia) możemy czytać następująco:

(lambda (x) (+ x 4))  
 ↑      ↑      ↑      ↑      ↑  
 procedura o argumencie x dodająca x i 4

Lambda-wyrażenie, jak każde wyrażenie, którego wartością jest procedura, może być użyte jako operator kombinacji takiej jak

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

lub, ogólniej, w dowolnym kontekście, w którym normalnie użylibyśmy nazwy procedury<sup>53</sup>.

### Tworzenie zmiennych lokalnych za pomocą let

Inne zastosowanie lambda-abstrakcji dotyczy tworzenia zmiennych lokalnych. Często w naszych procedurach potrzebujemy zmiennych lokalnych innych niż te związane jako parametry formalne. Założymy na przykład, że chcemy obliczyć funkcję

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

która moglibyśmy również wyrazić jako

$$\begin{aligned} a &= 1 + xy \\ b &= 1 - y \\ f(x, y) &= xa^2 + yb + ab \end{aligned}$$

Pisząc procedurę obliczającą  $f$ , chcielibyśmy jako zmienne lokalne wprowadzić nie tylko  $x$  i  $y$ , lecz także nazwy takich wartości pośrednich, jak  $a$  i  $b$ . Jednym ze sposobów na osiągnięcie tego jest zdefiniowanie pomocniczej procedury wiążącej zmienne lokalne:

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

<sup>53</sup> Nie wyglądałoby to tak strasznie i byłoby jaśniejsze dla osób uczących się Lispu, gdyby zamiast lambda użyto bardziej oczywistej nazwy, takiej jak `make-procedure`. Niestety, konwencja ta jest silnie zakorzeniona. Notację tę przejęto z rachunku  $\lambda$  — matematycznego formalizmu wprowadzonego przez matematyka i logika Alonzo Churcha [11]. Church opracował rachunek  $\lambda$ , aby dostarczyć ścisłych podstaw do badania pojęć funkcji i stosowania funkcji. Rachunek  $\lambda$  stał się podstawowym narzędziem do matematycznego badania semantyki języków programowania.

Oczywiście moglibyśmy użyć tu lambda-wyrażenia do opisania anonimowej procedury wiążącej nasze zmienne lokalne. Treść `f` staje się wówczas pojedynczym wywołaniem takiej procedury:

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

Konstrukcja ta jest tak bardzo użyteczna, że wprowadzono formę specjalną `let`, dzięki której jej użycie jest bardziej dogodne. Stosując `let`, moglibyśmy procedurę `f` zapisać tak:

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

Ogólna postać wyrażenia `let` jest następująca:

```
(let ((⟨v1⟩ ⟨e1⟩)
      (⟨v2⟩ ⟨e2⟩)
      :
      (⟨vn⟩ ⟨en⟩))
  ⟨treść⟩)
```

Można o tym myśleć następująco:

niech  $\langle v_1 \rangle$  ma wartość  $\langle e_1 \rangle$ ,  
 $\langle v_2 \rangle$  ma wartość  $\langle e_2 \rangle$ ,  
 $\vdots$   
 $\langle v_n \rangle$  ma wartość  $\langle e_n \rangle$ ,  
wówczas wynikiem jest  $\langle \text{treść} \rangle$

Pierwsza część wyrażenia `let` to lista par nazwa-wyrażenie. Gdy takie wyrażenie jest obliczane, z każdą nazwą jest wiązana wartość odpowiadającego mu wyrażenia. Przy obliczaniu wartości treści wyrażenia `let` nazwy te są zwią-

zane jako zmienne lokalne. Dzieje się to w ten sposób, że wyrażenie `let` jest interpretowane jako alternatywny sposób zapisu

```
((lambda ((v1) ... (vn))
  <treść>)
  (e1)
  :
  (en))
```

Aby wprowadzić zmienne lokalne, nie potrzeba w interpreterze żadnego nowego mechanizmu. Wyrażenia `let` są po prostu lukrem syntaktycznym pokrywającym użycie lambda-abstrakcji.

Z powyższej równoważności wynika, że zakresem zmiennych określonych w wyrażeniu `let` jest treść `let`. Stąd:

- `Let` pozwala na wiązanie zmiennych tak bardzo lokalnie względem miejsca ich użycia, jak to tylko możliwe. Jeśli na przykład wartością `x` jest 5, to wartością wyrażenia

```
(+ (let ((x 3))
      (+ x (* x 10)))
  x)
```

jest 38. W tym przypadku `x` w treści `let` jest równe 3, więc wartością wyrażenia `let` jest 33. Z kolei `x` będące drugim argumentem zewnętrznego dodawania jest nadal równe 5.

- Wartości zmiennych są obliczane poza `let`. Ma to znaczenie wtedy, gdy wyrażenia określające wartości zmiennych lokalnych zależą od zmiennych o takich samych nazwach co zmienne lokalne. Jeśli na przykład wartością `x` jest 2, to wyrażenie

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

ma wartość 12, gdyż wewnętrz treści `let` `x` jest równe 3, a `y` (równe zewnętrznemu `x` plus 2) jest równe 4.

Czasami, stosując definicje lokalne, możemy osiągnąć ten sam rezultat co przy użyciu `let`. Moglibyśmy na przykład zdefiniować powyższą procedurę `f` jako

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y)))
```

---

```
(+ (* x (square a))
  (* y b)
  (* a b)))
```

Wolimy jednak w takiej sytuacji używać `let`, a `define` wykorzystywać tylko do definiowania procedur lokalnych<sup>54</sup>.

### Ćwiczenie 1.34

Założmy, że zdefiniowaliśmy procedurę

```
(define (f g)
  (g 2))
```

Wówczas mamy

```
(f square)
4
(f (lambda (z) (* z (+ z 1))))
```

Co się stanie, jeśli (przekornie) poprosimy interpretera o obliczenie wartości kombinacji `(f f)`? Odpowiedź uzasadnij.

### 1.3.3. Procedury jako metody ogólne

W punkcie 1.1.4 wprowadziliśmy procedury złożone jako mechanizm tworzenia abstrakcji wzorców operacji liczbowych, dzięki czemu operacje te są niezależne od konkretnych liczb, których dotyczą. Wraz z procedurami wyższego rzędu, takimi jak przedstawiona w punkcie 1.3.1 procedura `integral`, pojawia się potężniejszy rodzaj abstrakcji: procedury wyrażające ogólne metody obliczeń, niezależne od konkretnych funkcji, których dotyczą. W tym punkcie omówimy dwa kolejne, bardziej złożone przykłady — ogólne metody znajdowania zer i punktów stałych funkcji — i pokażemy, jak wyrazić te metody bezpośrednio jako procedury.

#### Znajdowanie pierwiastków równań przez bisekcję

*Metoda bisekcji* (ang. *half-interval method*) jest prostą, acz potężną techniką służącą do znajdowania pierwiastków równań postaci  $f(x) = 0$ , gdzie  $f$  jest funkcją ciągłą. Pomyśl polega na tym, że jeżeli mamy dane takie punkty  $a$  i  $b$ , że  $f(a) < 0 < f(b)$ , to  $f$  musi mieć przynajmniej jedno zero między  $a$  i  $b$ .

---

<sup>54</sup> Zrozumienie definicji lokalnych w dostatecznym stopniu, żeby być pewnym, że znaczenie programu jest takie, jak zamierzaliśmy, wymaga bardziej dopracowanego modelu obliczania procesów niż przedstawiony w tym rozdziale. Subtelności te nie pojawiają się jednak przy definiowaniu procedur lokalnych. Wróćmy do tej kwestii w punkcie 4.1.6, gdy dowiemy się więcej o procesie obliczania.

Chcąc zlokalizować takie zero, oznaczmy przez  $x$  średnią z  $a$  i  $b$  i obliczmy  $f(x)$ . Jeśli  $f(x) > 0$ , to  $f$  musi mieć zero między  $a$  i  $x$ . Jeśli  $f(x) < 0$ , to  $f$  musi mieć zero między  $x$  i  $b$ . Postępując dalej w ten sposób, możemy określać coraz mniejsze przedziały, w których  $f$  musi mieć zero. Gdy uzyskamy dostatecznie mały przedział, przerywamy ten proces. Ponieważ przedział niepewności z każdym krokiem procesu zmniejsza się o połowę, więc wymagana liczba kroków jest rzędu  $\Theta(\log(L/T))$ , gdzie  $L$  jest długością początkowego przedziału, a  $T$  jest dopuszczalnym błędem (tzn. wielkością przedziału, który uważamy za „dostatecznie mały”). Oto procedura implementująca tę strategię:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                  (search f neg-point midpoint))
                ((negative? test-value)
                  (search f midpoint pos-point))
                (else midpoint)))))))
```

Zakładamy, że początkowo mamy daną funkcję  $f$  oraz punkty, w których przyjmuje ona wartość ujemną i dodatnią. Najpierw obliczamy punkt leżący pośrodku między dwoma danymi punktami. Następnie sprawdzamy, czy dany przedział jest dostatecznie mały, a jeśli tak, to wynikiem jest po prostu punkt środkowy. W przeciwnym razie sprawdzamy, jaka jest wartość  $f$  w tym punkcie. Jeśli jest ona dodatnia, to kontynuujemy proces dla nowego przedziału od punktu, w którym funkcja przyjmuje wartość ujemną, do punktu środkowego. Jeśli jest ona ujemna, to kontynuujemy proces dla przedziału od punktu środkowego do punktu, w którym funkcja przyjmuje wartość dodatnią. Wreszcie, istnieje możliwość, że badana wartość jest równa 0, a wówczas punkt środkowy jest szukanym pierwiastkiem.

Aby sprawdzić, czy końce przedziału leżą „dostatecznie blisko siebie”, możemy użyć procedury podobnej do procedury użytej w punkcie 1.1.7 przy obliczaniu pierwiastków kwadratowych<sup>55</sup>:

```
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

<sup>55</sup> Użyliśmy wówczas 0,001 jako przykładowej „małej” liczby określającej dopuszczalny błąd w obliczeniach. W prawdziwych obliczeniach określenie właściwej tolerancji błędu zależy od rozwiązywanego problemu, ograniczeń komputera i algorytmu. Są to często bardzo subtelne rozważania, wymagające pomocy specjalisty od analizy numerycznej lub innego rodzaju magika.

Bezpośrednie używanie procedury `search` jest niewygodne, gdyż możemy przez przypadek podać jej punkty, w których wartości  $f$  nie mają wymaganych znaków, a wówczas uzyskamy zły wynik. Zamiast tego będziemy korzystać z `search` za pośrednictwem poniższej procedury sprawdzającej, na którym z końców przedziału funkcja przyjmuje wartość ujemną, a na którym wartość dodatnia, i wywołującej odpowiednio procedurę `search`. Jeśli funkcja ma ten sam znak w obu danych punktach, to metoda bisekcji nie może być zastosowana — w takim przypadku procedura zgłasza błąd<sup>56</sup>.

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Wartości funkcji nie mają przeciwnych znaków:"
                  a b))))
```

W poniższym przykładzie metoda bisekcji została zastosowana do przybliżenia  $\pi$  jako leżącego między 2 i 4 pierwiastka równania  $\sin x = 0$ :

```
(half-interval-method sin 2.0 4.0)
3.14111328125
```

Oto inny przykład — użycie metody bisekcji do znalezienia pierwiastka równania  $x^3 - 2x - 3 = 0$  między 1 i 2:

```
(half-interval-method (lambda (x) (- (* x x x) (* 2 x) 3))
1.0
2.0)
1.89306640625
```

### Znajdowanie punktów stałych funkcji

Liczba  $x$  jest nazywana *punktem stałym* (ang. *fixed point*) funkcji  $f$ , jeśli  $x$  spełnia równanie  $f(x) = x$ . Dla niektórych funkcji  $f$  możemy zlokalizować punkt stały, zaczynając od pewnej wartości początkowej i stosując wielokrotnie  $f$ :

$f(x), f(f(x)), f(f(f(x))), \dots$

dopóki wartość nie przestanie się w istotny sposób zmieniać. Opierając się na tym pomyśle, możemy zbudować procedurę `fixed-point`, której argumen-

---

<sup>56</sup> Można to zrobić, korzystając z procedury `error`, która może mieć kilka argumentów wypisywanych jako komunikat błędu.

tami są funkcja i wartość początkowa, a wynikiem jest przybliżona wartość punktu stałego funkcji. Daną funkcję stosujemy wielokrotnie, dopóki różnica kolejnych dwóch wartości nie będzie mniejsza od pewnej ustalonej tolerancji:

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Możemy tej metody użyć na przykład do przybliżenia punktu stałego funkcji cosinus, zaczynając od 1 jako początkowego przybliżenia<sup>57</sup>:

```
(fixed-point cos 1.0)
.7390822985224023
```

Podobnie możemy znaleźć rozwiązanie równania  $y = \sin y + \cos y$ :

```
(fixed-point (lambda (y) (+ (sin y) (cos y)))
  1.0)
1.2587315962971173
```

Proces `fixed-point` przypomina proces, którego używaliśmy w punkcie 1.1.7 do znajdowania pierwiastków kwadratowych. Oba opierają się na pomyśle wielokrotnego polepszania przybliżenia, aż spełni ono pewne kryterium. W rzeczywistości możemy łatwo sformułować obliczanie pierwiastków kwadratowych jako poszukiwanie punktu stałego. Obliczenie pierwiastka kwadratowego z danej liczby  $x$  wymaga znalezienia takiego  $y$ , że  $y^2 = x$ . Przekształcając to równanie do równoważnej postaci  $y = x/y$ , rozpoznajemy, że szukamy punktu stałego funkcji<sup>58</sup>  $y \mapsto x/y$ , a zatem możemy spróbować obliczać pierwiastki kwadratowe jako

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
  1.0))
```

<sup>57</sup> Spróbuj tego w trakcie nudnego wykładu: ustaw kalkulator w tryb radianów i wielokrotnie naciskaj przycisk cos, aż uzyskasz punkt stały.

<sup>58</sup>  $\mapsto$  (czytaj: „przechodzi w”) to sposób matematyków na zapisywanie lambda-abstrakcji.  $y \mapsto x/y$  oznacza `(lambda(y) (/ x y))`, tzn. funkcję, której wartością w punkcie  $y$  jest  $x/y$ .

Niestety, ten proces poszukiwania punktu stałego nie jest zbieżny. Rozpatrzmy sytuację, gdy wartość początkowa jest równa  $y_1$ . Następne przybliżenie jest równe  $y_2 = x/y_1$ , a kolejne  $y_3 = x/y_2 = x/(x/y_1) = y_1$ . W rezultacie otrzymujemy pętlę nieskończoną, w której pojawiają się na przemian dwie wartości  $y_1$  i  $y_2$ , oscylujące wokół wyniku.

Jednym ze sposobów na opanowanie takich oscylacji jest zapobieżenie tak dużym zmianom przybliżeń. Skoro wynik zawsze leży między naszym przybliżeniem  $y$  a  $x/y$ , możemy więc jako kolejne przybliżenie wybrać liczbę, która nie różni się od  $y$  tak bardzo jak  $x/y$ , biorąc średnią z  $y$  i  $x/y$ . Wówczas kolejnym przybliżeniem po  $y$ , zamiast  $x/y$ , jest  $\frac{1}{2}(y + x/y)$ . Proces, w którym tworzymy taki ciąg przybliżeń, to po prostu proces poszukiwania punktu stałego przekształcenia  $y \mapsto \frac{1}{2}(y + x/y)$ :

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

(Zauważmy, że  $y = \frac{1}{2}(y + x/y)$  możemy w prosty sposób otrzymać z  $y = x/y$ ; wystarczy do obu stron równania dodać  $y$  i podzielić je przez 2).

Po wprowadzeniu tej modyfikacji procedura obliczania pierwiastków kwadratowych działa. W rzeczywistości, jeśli rozwiniemy definicje, to zobaczymy, że powstający ciąg przybliżeń pierwiastka kwadratowego jest dokładnie taki sam jak w przypadku naszej pierwszej procedury obliczającej pierwiastki kwadratowe, przedstawionej w punkcie 1.1.7. Takie podejście polegające na uśrednianiu kolejnych przybliżeń wyniku, nazywane *łumieniem przez uśrednienie* (ang. *average damping*), często pomaga w osiągnięciu zbieżności poszukiwań punktu stałego.

### Ćwiczenie 1.35

Pokaż, że złoty podział  $\phi$  (opisany w punkcie 1.2.2) jest punktem stałym przekształcenia  $x \mapsto 1 + 1/x$ , i użyj tego faktu do obliczenia  $\phi$  za pomocą procedury `fixed-point`.

### Ćwiczenie 1.36

Zmodyfikuj procedurę `fixed-point` tak, aby wypisywała powstający ciąg przybliżeń (używając procedur pierwotnych `newline` i `display` przedstawionych w ćwiczeniu 1.22). Następnie rozwiąż równanie  $x^x = 1000$ , znajdując punkt stały  $x \mapsto \log(1000)/\log(x)$ . (Użyj tutaj procedury pierwotnej `log` języka Scheme, obliczającej logarytm naturalny). Porównaj liczbę wykonywanych kroków z tłumieniem przez uśrednienie i bez niego. (Zwrć uwagę, że nie możesz uruchomić `fixed-point` z początkowym przybliżeniem równym 1, gdyż spowodowałoby to dzielenie przez  $\log(1) = 0$ ).

### Ćwiczenie 1.37

(a) Nieskończonym ułamkiem łańcuchowym (ang. *infinite continued fraction*) nazywamy wyrażenie postaci

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \dots}}}$$

Można na przykład pokazać, że nieskończony ułamek łańcuchowy, w którym wszystkie  $N_i$  i  $D_i$  są równe 1, jest równy  $1/\phi$ , gdzie  $\phi$  jest złotym podziałem (opisanym w punkcie 1.2.2). Jednym ze sposobów na przybliżenie nieskończonego ułamka łańcuchowego jest obcięcie jego rozwinięcia na określonej głębokości. Takie obcięcie — nazywane skończonym rozwinięciem ułamka łańcuchowego o głębokości  $k$  — jest postaci

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}$$

Przypuśćmy, że  $n$  i  $d$  są jednoargumentowymi procedurami, których wynikami (dla zadanego indeksu  $i$ ) są współczynniki ułamka łańcuchowego, odpowiednio  $N_i$  i  $D_i$ . Zdefiniuj taką procedurę `cont-frc`, że wynikiem (`cont-frc n d k`) jest skończone rozwinięcie ułamka łańcuchowego o głębokości  $k$ . Sprawdź swoją procedurę przez przybliżenie  $1/\phi$ , wywołując

```
(cont-frc (lambda (i) 1.0)
          (lambda (i) 1.0)
          k)
```

dla kolejnych liczb  $k$ . Jak duże musi być  $k$ , aby uzyskać przybliżenie z dokładnością do 4 miejsc dziesiętnych?

(b) Jeśli Twoja procedura `cont-frc` generuje proces rekurencyjny, to napisz taką, która generuje proces iteracyjny. Jeśli Twoja procedura `cont-frc` generuje proces iteracyjny, to napisz taką, która generuje proces rekurencyjny.

### Ćwiczenie 1.38

Szwajcarski matematyk Leonhard Euler opublikował w 1737 r. pracę naukową *De Fractionibus Continuis*, w której można znaleźć rozwinięcie  $e - 2$  w postaci ułamka łańcuchowego, gdzie  $e$  jest podstawą logarytmu naturalnego. W ułamku tym wszystkie  $N_i$  są równe 1, a  $D_i$  są kolejno równe: 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, .... Napisz program korzystający ze zdefiniowanej przez Ciebie w ćwiczeniu 1.37 procedury `cont-frc`, przybliżający  $e$  na podstawie rozwinięcia Eulera.

**Ćwiczenie 1.39**

W 1770 r. zostało opublikowane przez niemieckiego matematyka J. H. Lamberta przedstawienie funkcji tangens w postaci ułamka łaciuchowego:

$$\begin{aligned} \operatorname{tg} x = & \frac{x}{1 - \frac{x^2}{3 - \frac{x^2}{5 - \dots}}} \end{aligned}$$

gdzie  $x$  jest wielkościąkąta w radianach. Zdefiniuj procedurę (`tan-cf x k`) przybliżającą funkcję tangens na podstawie wzoru Lamberta. Jak w ćwiczeniu 1.37, k określa głębokość obliczanego rozwinięcia.

**1.3.4. Procedury jako wyniki**

Powysze przykłady pokazują, jak możliwość przekazywania procedur jako argumentów w znaczący sposób zwiększa siłę naszego języka programowania. Możemy osiągnąć jeszcze większą siłę wyrazu, tworząc procedury, których wyniki same są procedurami.

Możemy zilustrować ten pomysł, rozpatrując ponownie przykład obliczania punktu stałego, opisany na końcu punktu 1.3.3. Wychodząc od obserwacji, że  $\sqrt{x}$  jest punktem stałym funkcji  $y \mapsto x/y$ , sformułowaliśmy nową wersję procedury obliczającej pierwiastki kwadratowe za pomocą wyznaczania punktu stałego funkcji. Następnie użyliśmy tłumienia przez uśrednienie, aby proces przybliżania był zbieżny. Tłumienie przez uśrednienie jest samo w sobie użyteczną i ogólną techniką. Mianowicie, mając daną funkcję  $f$ , rozważamy funkcję, której wartość w punkcie  $x$  jest równa średniej z  $x$  i  $f(x)$ .

Pojęcie tłumienia przez uśrednienie możemy wyrazić następującą procedurą:

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

Procedura `average-damp` ma jeden argument  $f$  będący procedurą, a jej wynikiem jest procedura (tworzona przez `lambda-abstrakcję`), której wynikiem dla liczby  $x$  jest średnia z  $x$  i  $(f x)$ . Stosując na przykład `average-damp` do procedury `square`, otrzymujemy procedurę, której wartością dla danej liczby  $x$  jest średnia z  $x$  i  $x^2$ . Stosując procedurę otrzymaną w wyniku do liczby 10, otrzymujemy średnią z 10 i 100, czyli 55<sup>59</sup>:

```
((average-damp square) 10)
```

55

<sup>59</sup> Zauważmy, że mamy tu do czynienia z kombinacją, której operator sam też jest kombinacją. Ćwiczenie 1.4 pokazało już, że tworzenie takich kombinacji jest możliwe, ale był to tylko modelowy przykład. Teraz, gdy możemy zastosować procedurę otrzymaną jako wynik procedury wyższego rzędu, okazuje się, że takie kombinacje są naprawdę potrzebne.

Używając `average-damp`, możemy w następujący sposób ponownie sformułować procedurę pierwiastkowania:

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

Zauważmy, jak takie sformułowanie uwidacznia trzy pojęcia pojawiające się w tej procedurze: znajdowanie punktu stałego, tłumienie przez uśrednienie i funkcję  $y \mapsto x/y$ . Porównanie takiego sformułowania metody pierwiastkowania z pierwotną wersją, podaną w punkcie 1.1.7, może być pouczające. Pamiętajmy, że obie procedury wyrażają te same procesy, i zauważmy, o ile jaśniejsza staje się nasza koncepcja, gdy proces jest wyrażony za pomocą wymienionych abstrakcji. Ogólnie mówiąc, jest wiele sposobów sformułowania procesu w postaci procedury. Doświadczani programiści wiedzą, jak formułować procedury w sposób przejrzysty i ukazujący użyteczne elementy procesu jako osobne całości, które mogą być ponownie użyte w innych zastosowaniach. Oto prosty przykład ponownego użycia — zauważmy, że pierwiastek sześcienny z  $x$  jest punktem stałym funkcji  $y \mapsto x/y^2$ , możemy więc natychmiast uogólnić naszą procedurę obliczania pierwiastków kwadratowych, uzyskując procedurę obliczającą pierwiastki sześcienne<sup>60</sup>:

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

### Metoda Newtona

Kiedy w punkcie 1.1.7 przedstawiliśmy po raz pierwszy procedurę obliczającą pierwiastki kwadratowe, wspomnieliśmy, że jest to szczególny przypadek *metody Newtona*. Jeśli  $x \mapsto g(x)$  jest funkcją różniczkowalną [o niezerowej pochodnej, przynajmniej w interesujących nas punktach; przyp. tłum.], to rozwiązanie równania  $g(x) = 0$  jest punktem stałym funkcji  $x \mapsto f(x)$ , gdzie

$$f(x) = x - \frac{g(x)}{g'(x)}$$

a  $g'(x)$  jest pochodną funkcji  $g$  w punkcie  $x$ . Metoda Newtona polega na zastosowaniu metody punktu stałego, którą widzieliśmy powyżej, do przybliżonego rozwiązania równania poprzez znalezienie punktu stałego funkcji  $f$ <sup>61</sup>. Dla

<sup>60</sup> Dalsze uogólnienie można znaleźć w ćwiczeniu 1.45.

<sup>61</sup> W podręcznikach podstaw analizy przy opisywaniu metody Newtona mówi się zazwyczaj o ciągu przybliżeń postaci  $x_{n+1} = x_n - g(x_n)/g'(x_n)$ . Posługiwanie się językiem, w którym możemy używać pojęć procesu i punktu stałego, upraszcza opis tej metody.

wielu funkcji  $g$  przy odpowiednio dobrym pierwszym przybliżeniu wyniku  $x$  metoda Newtona bardzo szybko dąży do rozwiązania  $g(x) = 0$ <sup>62</sup>.

Aby zaimplementować metodę Newtona w postaci procedury, musimy najpierw wyrazić pojęcie pochodnej (ang. *derivative*). Zwróćmy uwagę, że „pochodna”, tak jak tłumienie przez średnienie, jest czymś, co przekształca jedną funkcję w drugą. Na przykład pochodną funkcji  $x \mapsto x^3$  jest funkcja  $x \mapsto 3x^2$ . Ogólnie, jeśli  $g$  jest funkcją, a  $dx$  jest małą liczbą, to pochodna  $g'$  funkcji  $g$  jest funkcją, której wartość dla każdego punktu  $x$  jest określona przez granicę:

$$g'(x) = \lim_{dx \rightarrow 0} \frac{g(x + dx) - g(x)}{dx}$$

Tak więc możemy przybliżyć pojęcie pochodnej (ustalając wartość  $dx$  na, powiedzmy, 0,00001) za pomocą procedury

```
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
        dx)))
```

wraz z definicją

```
(define dx 0.00001)
```

Jak *average-damp*, również *deriv* jest procedurą, której zarówno argument, jak i wynik są procedurami. Aby na przykład przybliżyć wartość pochodnej funkcji  $x \mapsto x^3$  w punkcie 5 (równą dokładnie 75), możemy obliczyć

```
(define (cube x) (* x x x))
((deriv cube) 5)
75.00014999664018
```

Za pomocą *deriv* możemy wyrazić metodę Newtona jako proces przybliżania punktu stałego:

```
(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

<sup>62</sup> Metoda Newtona nie zawsze jest zbieżna do wyniku, ale można pokazać, że w sprzyjających warunkach każdy krok podwaja liczbę miejsc dziesiętnych dokładności przybliżenia wyniku. W takich przypadkach metoda Newtona będzie zbieżna do wyniku dużo szybciej niż metoda bisekcji.

Procedura `newton-transform` wyraża formułę przedstawioną na początku tego punktu, a `newtons-method` jest w łatwy sposób zdefiniowana za jej pomocą. Argumentami `newtons-method` są: procedura obliczająca funkcję, której zera szukamy, oraz początkowa wartość przybliżenia. Chcąc na przykład znaleźć pierwiastek kwadratowy z  $x$ , możemy użyć metody Newtona do wyznaczenia zera funkcji  $y \mapsto y^2 - x$ , zaczynając od początkowego przybliżenia równego 1<sup>63</sup>. Daje nam to jeszcze jedną postać procedury pierwiastkowania:

```
(define (sqrt x)
  (newtons-method (lambda (y) (- (square y) x))
    1.0))
```

### Abstrakcje i procedury jako pierwszorzędne elementy języka

Dotychczas widzieliśmy dwa sposoby wyrażenia obliczania pierwiastków kwadratowych jako szczególnych przypadków ogólniejszych metod: poszukiwania punktu stałego i metody Newtona. Ponieważ metoda Newtona była sama wyrażona jako proces obliczania punktu stałego, więc faktycznie widzieliśmy dwa sposoby obliczania pierwiastków kwadratowych jako punktów stałych. Każda z tych metod ma na początku daną funkcję, przekształca ją i znajduje punkt stały przekształconej funkcji. Możemy wyrazić tę ogólną ideę w postaci procedury:

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

Ta bardzo ogólna procedura ma trzy argumenty: procedurę `g` obliczającą pewną funkcję, procedurę `transform` przekształcającą `g` i wartość początkową `guess`. Jej wynikiem jest punkt stały przekształconej funkcji.

Używając tej abstrakcji, możemy ponownie wyrazić pierwszą metodę obliczania pierwiastków kwadratowych, przedstawioną w tym punkcie (polegającą na szukaniu punktu stałego, stłumionego przez uśrednienie, przekształcenia  $y \mapsto x/y$ ), jako przypadek takiej metody ogólnej:

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (/ x y))
    average-damp
    1.0))
```

Podobnie możemy wyrazić drugą metodę obliczania pierwiastków kwadratowych, przedstawioną w tym punkcie (przypadek szczególny metody Newtona,

<sup>63</sup> W przypadku znajdowania pierwiastków kwadratowych metoda Newtona bardzo szybko dąży do poprawnego wyniku, dla dowolnego punktu początkowego [różnego od  $-\frac{dx}{2}$ ; przyp. tłum.].

polegający na znajdowaniu punktu stałego funkcji  $y \mapsto y^2 - x$  przekształconej zgodnie z metodą Newtona):

```
(define (sqrt x)
  (fixed-point-of-transform (lambda (y) (- (square y) x))
    newton-transform
    1.0))
```

Podrozdział 1.3 rozpoczęliśmy od stwierdzenia, że procedury złożone są podstawowym mechanizmem abstrakcji, ponieważ umożliwiają wprowadzenie do naszego języka programowania ogólnych metod obliczeniowych, jako jawnych elementów tego języka. Widzieliśmy, jak procedury wyższych rzędów pozwalają na operowanie tymi ogólnymi metodami i tworzenie dalszych abstrakcji.

Jako programiści, powinniśmy być czujni i rozpoznawać abstrakcje leżące u podstawa naszych programów, korzystać z nich i uogólniając je, tworzyć jeszcze potężniejsze abstrakcje. Nie oznacza to, że zawsze powinniśmy pisać programy w sposób możliwie najbardziej abstrakcyjny; wytrawni programiści potrafią dobrać poziom abstrakcji odpowiedni do zadania. Jest jednak ważne, aby umieć myśleć, posługując się abstrakcjami, i być gotowym do zastosowania ich w nowych kontekstach. Znaczenie procedur wyższych rzędów polega na tym, że umożliwiają one przedstawianie abstrakcji wprost jako elementów w naszym języku programowania, dzięki czemu możemy obchodzić się z nimi jak z pozostałymi elementami języka programowania.

Ogólnie mówiąc, języki programowania nakładają ograniczenia na sposoby, w jakie możemy manipulować ich elementami. Te elementy, których dotyczą najmniejsze ograniczenia, są nazywane *pierwszorzędnymi*. Oto niektóre z „praw i przywilejów” elementów pierwszorzędnych<sup>64</sup>:

- Mogą być wartościami zmiennych.
- Mogą być argumentami procedur.
- Mogą być wynikami procedur.
- Mogą być elementami struktur danych<sup>65</sup>.

Lisp, w odróżnieniu od innych powszechnie stosowanych języków programowania, w pełni nadaje procedurom status pierwszorzędnych elementów języka programowania. Powoduje to, że stworzenie efektywnej implementacji staje

<sup>64</sup> Pojęcie pierwszorzędnego statusu elementów języków programowania wprowadził brytyjski informatyk Christopher Strachey (1916–1975).

<sup>65</sup> Przykłady na to zobaczymy po wprowadzeniu w rozdziale 2 struktur danych.

się trudnym zadaniem, ale za to siła wyrazu, jaką dzięki temu można uzyskać, jest ogromna<sup>66</sup>.

### Ćwiczenie 1.40

Zdefiniuj procedurę `cubic`, której można użyć razem z procedurą `newtons-method` w następujący sposób do przybliżenia zer wielomianu  $x^3 + ax^2 + bx + c$ :

```
(newtons-method (cubic a b c) 1)
```

### Ćwiczenie 1.41

Zdefiniuj procedurę `double`, której argumentem jest procedura jednoargumentowa, a wynikiem procedura polegająca na dwukrotnym zastosowaniu procedury będącej jej argumentem. Przykładowo, jeśli `inc` jest procedurą dodającą 1 do swojego argumentu, to `(double inc)` powinno być procedurą, która dodaje 2. Jaki jest wynik wywołania

```
((double (double double)) inc) 5)
```

### Ćwiczenie 1.42

Niech  $f$  i  $g$  będą dwiema funkcjami jednoargumentowymi. *Złożenie*  $f$  z  $g$  jest zdefiniowane jako funkcja  $x \mapsto f(g(x))$ . Zdefiniuj procedurę `compose` implementującą złożenie funkcji. Przykładowo, jeśli `inc` jest procedurą dodającą 1 do swojego argumentu, to

```
((compose square inc) 6)
```

49

### Ćwiczenie 1.43

Jeśli  $f$  jest funkcją określoną na liczbach, a  $n$  jest dowolną dodatnią liczbą całkowitą, to  $n$ -krotnym złożeniem funkcji  $f$  nazywamy funkcję, której wartością jest wynik  $n$ -krotnego zastosowania funkcji  $f$ :  $x \mapsto f(f(\dots(f(x))\dots))$ . Przykładowo, jeśli  $f$  jest funkcją  $x \mapsto x + 1$ , to  $n$ -krotne złożenie  $f$  jest funkcją  $x \mapsto x + n$ . Jeśli  $f$  jest operacją podnoszenia do kwadratu, to  $n$ -krotne złożenie  $f$  jest funkcją podnoszącą swój argument do potęgi  $2^n$ . Napisz procedurę `repeated`, której argumentami są procedura obliczająca  $f$  i dodatnia liczba całkowita  $n$ , a wynikiem jest procedura obliczająca  $n$ -krotne złożenie  $f$ . Z Twojej procedury powinno się korzystać w następujący sposób:

```
((repeated square 2) 5)
```

625

Wskazówka: Może Ci się przydać procedura `compose` z ćwiczenia 1.42.

<sup>66</sup> Większość kosztów implementacyjnych związanych z procedurami pierwszorzędnymi wynika stąd, że pozwalamy na to, aby procedury były wynikami działania innych procedur. Wymaga to zarezerwowania pamięci przeznaczonej na zmienne wolne występujące w procedurze — nawet wówczas, gdy nie jest ona wykonywana. W implementacji języka Scheme, którą będziemy badali w podrozdziale 4.1, zmienne te są przechowywane w środowisku procedury.

### Ćwiczenie 1.44

Wygładzanie (ang. *smoothing*) funkcji jest ważnym pojęciem występującym w przetwarzaniu sygnałów. Jeśli  $f$  jest funkcją, a  $dx$  jest małą liczbą, to wygładzona funkcja  $f$  jest funkcją, której wartość w punkcie  $x$  jest średnią z  $f(x - dx)$ ,  $f(x)$  i  $f(x + dx)$ . Napisz procedurę `smooth`, której argumentem jest procedura obliczająca  $f$ , a wynikiem jest procedura obliczająca wygładzoną funkcję  $f$ . Czasami warto wielokrotnie wygładzać funkcję (tzn. wygładzać wygładzoną funkcję itd.), otrzymując *n-krotnie wygładzoną funkcję*. Pokaż, jak można wygenerować *n-krotnie wygładzoną funkcję* dowolnej danej funkcji, używając procedury `smooth` i procedury `repeated` (z ćwiczenia 1.43).

### Ćwiczenie 1.45

W punkcie 1.3.3 widzieliśmy, że naiwna próba obliczania pierwiastków kwadratowych poprzez szukanie punktu stałego funkcji  $y \mapsto x/y$  nie była zbieżna, i poprawiliśmy ją za pomocą tłumienia przez uśrednienie. Ta sama metoda działa w przypadku znajdowania pierwiastków sześciennych jako punktów stałych tłumionej przez uśrednienie funkcji  $y \mapsto x/y^2$ . Niestety, nie zadziała ona dla pierwiastków czwartego stopnia — jednokrotne tłumienie przez uśrednienie nie wystarcza, aby poszukiwanie punktu stałego dla  $y \mapsto x/y^3$  było zbieżne. Z kolei, jeśli zastosujemy dwukrotne tłumienie przez uśrednienie (tzn. stłumimy przez uśrednienie stłumioną przez uśrednienie funkcję  $y \mapsto x/y^3$ ), to poszukiwanie punktu stałego będą zbieżne. Poeksperymentuj i ustal, ilekrotnie tłumienie przez uśrednienie jest potrzebne do obliczenia pierwiastka  $n$ -go stopnia jako punktu stałego (wielokrotnie tłumionej przez uśrednienie) funkcji  $y \mapsto x/y^{n-1}$ . Wyników użyj do zaimplementowania prostej procedury obliczającej pierwiastek  $n$ -go stopnia za pomocą procedur `fixed-point`, `average-damp` i `repeated` (z ćwiczenia 1.43). Możesz założyć, że wszystkie potrzebne operacje arytmetyczne są dostępne jako procedury pierwotne.

### Ćwiczenie 1.46

Kilka metod numerycznych opisanych w tym rozdziale, to szczególnie przypadki niezmiernie ogólnej strategii obliczeniowej znanej jako *metoda kolejnych przybliżeń* (ang. *iterative improvement*). Metoda ta polega na tym, że aby coś obliczyć, zaczynamy od pewnego początkowego przybliżenia wyniku, sprawdzamy, czy przybliżenie jest dostatecznie dobre, i jeśli nie jest, to polepszamy je i kontynuujemy ten proces, używając polepszonego przybliżenia. Napisz procedurę `iterative-improve`, której argumentami są dwie procedury: metoda stwierdzania, czy przybliżenie jest dostatecznie dobre, i metoda polepszania przybliżenia. Wynikiem procedury `iterative-improve` powinna być procedura, której argumentem jest pierwsze przybliżenie i która polepsza je tak długo, aż będzie dostatecznie dobre. Przepisz procedury `sqrt` z punktu 1.1.7 i `fixed-point` z punktu 1.3.3, używając procedury `iterative-improve`.

# 2

## Budowanie abstrakcji za pomocą danych

Dochodzimy teraz do decydującego kroku matematycznej abstrakcji: możemy zapomnieć, co oznaczają symbole.

...[Matematyk] nie musi być bezczynny; jest wiele operacji, które może wykonywać na tych symbolach, nie wiedząc nawet, co one oznaczają.

Hermann Weyl, *The Mathematical Way of Thinking*

W rozdziale 1 skoncentrowaliśmy się na procesach obliczeniowych i na roli, jaką odgrywają procedury w konstruowaniu programów. Zobaczyliśmy, jak używa się danych pierwotnych (liczb) i operacji pierwotnych (operacji arytmetycznych), jak łączy się procedury w procedury złożone za pomocą złożenia, wyrażeń warunkowych i parametrów oraz jak tworzy się abstrakcje proceduralne za pomocą `define`. Dowiedzieliśmy się, że procedurę można uważać za wzorzec, według którego następuje lokalny rozwój procesu obliczeniowego, oraz sklasyfikowaliśmy, rozważyliśmy i dokonaliśmy prostych analiz algorytmicznych kilku powszechnie spotykanych wzorców procesów realizowanych przez procedury. Przekonaliśmy się również, że procedury wyższych rzędów zwiększą siłę wyrazu naszego języka, pozwalając nam operować, a tym samym posługiwać się w naszych rozumowaniach, ogólnymi metodami obliczeń. Wszystko to w dużym stopniu stanowi istotę programowania.

W niniejszym rozdziale przyjrzymy się bardziej złożonym danym. Wszystkie procedury z rozdziału 1 operowały na prostych danych liczbowych, a proste dane nie są wystarczające w przypadku wielu problemów, które chcielibyśmy rozwiązywać za pomocą obliczeń. Programy służą zazwyczaj do modelowania złożonych zjawisk, a wówczas, żeby zamodelować rzeczywiste zjawiska o wiele aspektach, najczęściej trzeba tworzyć obiekty obliczeniowe złożone z wielu części. Tak więc, podczas gdy w rozdziale 1 nasza uwaga była skupiona na budowaniu abstrakcji za pomocą łączenia procedur prostych w złożone, w tym rozdziale zwracamy się ku innemu kluczowemu aspektowi każdego języka programowania: średkiem, jakie udostępnia on do budowania abstrakcji za pomocą łączenia obiektów danych w *dane złożone*.

Po co nam w języku programowania dane złożone? Z tego samego powodu, dla którego potrzebujemy procedur złożonych: aby podnieść poziom pojęciowy, na którym konstruujemy nasze programy, aby zwiększyć modularność naszych

konstrukcji i siłę wyrazu naszego języka. Jak możliwość definiowania procedur pozwala nam na zajmowanie się procesami na poziomie pojęciowym wyższym niż poziom pierwotnych operacji języka, tak możliwość konstruowania złożonych obiektów danych pozwala nam na zajmowanie się danymi na poziomie pojęciowym wyższym niż poziom pierwotnych obiektów danych języka.

Rozważmy zadanie polegające na skonstruowaniu systemu wykonującego operacje arytmetyczne na liczbach wymiernych. Możemy sobie wyobrazić operację `add-rat` o dwóch argumentach będących liczbami wymiernymi, której wynikiem jest ich suma. Posługując się danymi prostymi, możemy wyobrazić sobie liczbę wymierną jako dwie liczby całkowite stanowiące odpowiednio licznik i mianownik. Tak więc moglibyśmy skonstruować program, w którym każda liczba wymienna byłaby reprezentowana przez dwie liczby całkowite (licznik i mianownik) i w którym operacja `add-rat` byłaby zaimplementowana przez dwie procedury (jedną, której wynikiem jest licznik sumy, i drugą, której wynikiem jest mianownik sumy). Byłoby to jednak bardzo niewygodne, gdyż musielibyśmy jawnie śledzić, który licznik odpowiada któremu mianownikowi. W systemie przeznaczonym do wykonywania wielu operacji na wielu liczbach wymiernych „księgowanie” takiej liczby szczegółów znacznie zaśmiecałoby programy, nie mówiąc już o naszych głowach. Byłoby znacznie lepiej, gdybyśmy mogli „skleić ze sobą” licznik i mianownik w parę — *złożony obiekt danych* (ang. *compound data object*) — którą nasze programy mogłyby traktować jako pojęciową całość, tak jak sami traktujemy liczby wymierne.

Zastosowanie danych złożonych umożliwia również zwiększenie modularności programów. Gdybyśmy mogli bezpośrednio operować na obiektach reprezentujących liczby wymierne same w sobie, to moglibyśmy oddzielić część naszego programu dotyczącą liczb wymiernych jako takich od szczegółów reprezentacji liczb wymiernych jako par liczb całkowitych. Ogólna metoda odzielania od siebie części programów dotyczących tego, jak obiekty danych są reprezentowane, od części programów dotyczących tego, jak te obiekty są używane, jest potężną techniką nazywaną *abstrakcją danych* (ang. *data abstraction*). Zobaczmy, jak abstrakcja danych czyni programy łatwiejszymi do konstruowania, pielęgnowania i modyfikowania.

Zastosowanie danych złożonych prowadzi do prawdziwego wzrostu siły wyrazu naszego języka programowania. Rozważmy pomysł utworzenia „kombinacji liniowej”  $ax + by$ . Moglibyśmy chcieć zapisać procedurę, która przyjmowałaby jako argumenty  $a$ ,  $b$ ,  $x$  i  $y$ , zaś w wyniku dawała wartość  $ax + by$ . Jeśli argumenty są liczbami, nie przedstawia to żadnych trudności, gdyż możemy od razu zdefiniować procedurę

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

Przypuśćmy jednak, że interesują nas nie tylko liczby. Przypuśćmy, że chcielibyśmy wyrazić za pomocą procedury koncepcję tworzenia kombinacji liniowej zawsze, kiedy są określone dodawanie i mnożenie — czy to liczb wymiernych, czy liczb zespolonych, czy wielomianów, czy też czegokolwiek innego. Moglibyśmy to zapisać w postaci następującej procedury:

```
(define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

gdzie `add` i `mul` nie są procedurami pierwotnymi `+` i `*`, ale raczej czymś bardziej złożonym, co będzie wykonywało odpowiednie operacje na dowolnego rodzaju danych, przekazywanych przez nas jako argumenty `a`, `b`, `x` i `y`. Zasadnicze znaczenie ma tu fakt, że procedura `linear-combination` musi wiedzieć o `a`, `b`, `x` i `y` jedynie to, że procedury `add` i `mul` będą wykonywać odpowiednie operacje. Z punktu widzenia procedury `linear-combination` nie ma znaczenia, czym są `a`, `b`, `x` i `y`, a jeszcze bardziej nie ma znaczenia to, jak mogą one być reprezentowane za pomocą bardziej elementarnych danych. Ten sam przykład pokazuje, dlaczego jest tak ważne, aby nasz język programowania dawał możliwość bezpośredniego operowania obiektami złożonymi — bez tego taka procedura jak `linear-combination`, nie znając szczegółowej struktury swoich argumentów, nie mogłaby przekazać ich procedurom `add` i `mul`<sup>1</sup>.

Rozdział ten zaczniemy od zaimplementowania, wspomnianej wcześniej, arytmetyki liczb wymiernych. Uzyskamy w ten sposób tło, na którym będziemy omawiać dane złożone i abstrakcję danych. Jak w przypadku procedur złożonych, główną kwestią, którą należy się zająć, jest abstrakcja jako technika radzenia sobie ze złożonością. Zobaczmy, jak abstrakcja danych umożliwia nam wznoszenie między różnymi częściami programu *barier abstrakcji* (ang. *abstraction barriers*).

Dowieemy się, że kluczem do tworzenia danych złożonych jest to, żeby język programowania dostarczał pewnego rodzaju „kleju” umożliwiającego łączenie obiektów danych w celu utworzenia obiektów bardziej złożonych. Jest wiele możliwych rodzajów takiego kleju. Odkryjemy nawet, jak tworzyć dane złożone, nie korzystając z żadnych specjalnych „operacji na danych”, a używając tylko procedur. W jeszcze większym stopniu zatrze to rozróżnienie między

<sup>1</sup> Możliwość bezpośredniego operowania na procedurach daje nam analogiczny wzrost siły wyrazu języka programowania. W punkcie 1.3.1 wprowadziliśmy na przykład procedurę `sum`, której argumentem jest procedura `term` i która oblicza sumę wartości procedury `term` w określonym przedziale. Kluczowe do zdefiniowania `sum` jest to, abyśmy mogli o takich procedurach jak `term` mówić jako o całościach samych w sobie, nie zajmując się tym, jak mogą być one wyrażone za pomocą prostszych operacji. Rzeczywiście, gdybyśmy nie posługiwali się pojęciem „procedury”, to wątpliwe, czy kiedykolwiek pomyślelibyśmy chociaż o możliwości zdefiniowania takiej operacji jak `sum`. Co więcej, jak długo interesuje nas wykonywanie dodawania, tak długo szczegółowo zapisu `term` za pomocą prostszych operacji są nieistotne.

„procedurami” i „danymi”, które i tak już pod koniec rozdziału 1 stało się niewyraźne. Zbadamy też pewne typowe sposoby reprezentowania ciągów i drzew. Przy rozważaniu danych złożonych kluczowym pomysłem jest wykorzystanie własności *domknięcia* (ang. *closure*) — polegającej na tym, że klej, którego używamy do łączenia obiektów danych, powinien umożliwiać łączenie nie tylko pierwotnych, lecz także złożonych obiektów danych. Inny kluczowy pomysł polega na tym, że złożone obiekty danych mogą być używane jako *konwencjonalne interfejsy* (ang. *conventional interfaces*) do łączenia modułów programu na najrozmaitsze sposoby. Niektóre z tych pomysłów zilustrujemy, przedstawiając prosty język graficzny korzystający z własności domknięcia.

Następnie zwiększymy możliwości reprezentacyjne naszego języka, wprowadzając *wyrażenia symboliczne* (ang. *symbolic expressions*) — dane, których podstawowymi elementami mogą być dowolne symbole, a nie tylko liczby. Poznamy różne możliwe sposoby reprezentowania zbiorów obiektów. Dowiemy się, że tak jak dana funkcja liczbową może być obliczana przez wiele różnych procesów obliczeniowych, tak samo dana struktura danych może być na wiele sposobów reprezentowana za pomocą prostszych obiektów, a wybór reprezentacji danych może mieć znaczący wpływ na wymagania czasowe i pamięciowe procesów operujących na tych danych. Rozpatrzymy te kwestie na przykładzie różniczkowania symbolicznego, reprezentowania zbiorów i kodowania informacji.

W dalszej kolejności zajmiemy się problemem pracy z danymi, które mogą być w różny sposób reprezentowane przez różne części programu. Prowadzi to do potrzeby zaimplementowania *operacji ogólnych* (ang. *generic operations*), które muszą sobie radzić z wieloma różnymi rodzajami danych. Zachowanie modularności przy jednoczesnej obecności operacji ogólnych wymaga mocniejszych barier abstrakcji niż te, które można wznieść jedynie za pomocą prostej abstrakcji danych. W szczególności wprowadzimy technikę *programowania sterowanego danymi* (ang. *data-directed programming*), pozwalającą na niezależne konstruowanie poszczególnych reprezentacji danych, a następnie na ich *addytywne łączenie* (tzn. łączenie bez potrzeby modyfikowania ich). Na zakończenie rozdziału, aby zilustrować moc takiego podejścia do konstruowania systemów, zastosujemy zdobytą wiedzę do zaimplementowania pakietu wykonującego symboliczne operacje arytmetyczne na wielomianach, w którym współczynnikami wielomianów mogą być liczby całkowite, liczby wymierne, liczby zespolone, a nawet inne wielomiany.

## 2.1. Wprowadzenie do abstrakcji danych

W punkcie 1.1.8 zauważyliśmy, że procedurę będącą elementem służącym do budowy bardziej złożonych procedur możemy traktować nie tylko jako zestaw poszczególnych operacji, lecz także jako abstrakcję proceduralną. Oznacza to,

że szczegóły tego, jak procedura była zaimplementowana, mogłyby być pominięte, a samą procedurę można by było zastąpić dowolną inną procedurą o takim samym ogólnym zachowaniu. Innymi słowy, poprzez abstrakcję możemy oddzielić sposób, w jaki procedura będzie używana, od szczegółów jej implementacji za pomocą prostszych procedur. Analogiczne pojęcie dotyczące danych złożonych jest nazywane *abstrakcją danych*. Abstrakcja danych to technika pozwalająca na oddzielenie tego, jak używamy złożonych obiektów danych, od szczegółów tego, jak są one zbudowane z prostszych obiektów danych.

Podstawowy pomysł w abstrakcji danych polega na tym, aby programom korzystającym ze złożonych obiektów danych nadawać taką strukturę, żeby operowały one na „danych abstrakcyjnych”. Oznacza to, że nasze programy powinny używać danych w taki sposób, żeby nie trzeba było zakładać o tych danych nic, co nie jest bezwzględnie konieczne do wykonania danego zadania. Równocześnie „konkretna” reprezentacja danych jest definiowana niezależnie od programów korzystających z tych danych. Interfejsem między tymi dwiema częściami systemu jest zestaw procedur, nazywanych *selektorami* (ang. *selectors*) i *konstruktorami* (ang. *constructors*), implementujących dane abstrakcyjne za pomocą ich konkretnej reprezentacji. Zilustrujemy tę technikę, konstruując przykładowy zestaw procedur operujących na liczbach wymiernych.

### 2.1.1. Przykład: operacje arytmetyczne na liczbach wymiernych

Przypuśćmy, że potrzebne nam są operacje arytmetyczne na liczbach wymiernych. Chcemy mieć możliwość dodawania, odejmowania, mnożenia i dzielenia, a także sprawdzania, czy dwie liczby wymierne są równe.

Założymy na początek, że potrafimy konstruować liczby wymierne z liczników i mianowników. Założymy też, że mając daną liczbę wymierną, potrafimy wydobyć z niej (za pomocą selektorów) jej licznik i mianownik. Przyjmijmy jeszcze, że konstruktor i selektory są dostępne w postaci następujących procedur:

- `(make-rat <n> <d>)` — jej wynikiem jest liczba wymierna o liczniku równym liczbie całkowitej `<n>` i mianowniku równym liczbie całkowitej `<d>`;
- `(numer <x>)` — jej wynikiem jest licznik liczby wymiernej `<x>`;
- `(denom <x>)` — jej wynikiem jest mianownik liczby wymiernej `<x>`.

Używamy tu potężnej strategii syntezy: tzw. *pobożnych życzeń* (ang. *wishful thinking*). Nie powiedzieliśmy jeszcze, jak liczby wymierne są reprezentowane ani jak procedury `numer`, `denom` i `make-rat` powinny być zaimplementowane. Mimo to, gdybyśmy mieli te trzy procedury, moglibyśmy wówczas dodawać, odejmować, mnożyć, dzielić i porównywać liczby wymierne, korzystając z na-

stępujących tożsamości:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1d_2 + n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1d_2 - n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1n_2}{d_1d_2}$$

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1d_2}{d_1n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \quad \text{wtedy i tylko wtedy, gdy} \quad n_1d_2 = n_2d_1$$

Możemy te reguły wyrazić w postaci poniższych procedur:

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y)))))

(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y)))))

(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y)))))

(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y)))))

(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Mamy teraz zdefiniowane, za pomocą konstruktora `make-rat` oraz selektorów `numer` i `denom`, operacje na liczbach wymiernych. Jednakże samego konstruktora i selektorów jeszcze nie zdefiniowaliśmy. Potrzebny nam jest do tego jakiś sposób sklejenia ze sobą licznika i mianownika liczby wymiernej.

### Par

Aby umożliwić nam zaimplementowanie konkretnej reprezentacji abstrakcji danych, nasz język udostępnia strukturę złożoną nazywaną *par* (ang. *pair*). Parę można utworzyć za pomocą procedury pierwotnej `cons`. Procedura ta ma dwa argumenty, a jej wynikiem jest złożony obiekt danych zawierający, jako

swoje części, te dwa argumenty. Mając daną parę, możemy z niej wydobyć jej części za pomocą procedur pierwotnych `car` i `cdr`<sup>2</sup>. Tak więc możemy korzystać z `cons`, `car` i `cdr` w następujący sposób:

```
(define x (cons 1 2))

(car x)
1

(cdr x)
2
```

Zauważmy, że para jest obiektem danych, który można nazwać i którym możemy się posługiwać tak jak obiektem pierwotnym. Co więcej, za pomocą `cons` możemy tworzyć pary, których elementami są również pary itd.:

```
(define x (cons 1 2))

(define y (cons 3 4))

(define z (cons x y))

(car (car z))
1

(car (cdr z))
3
```

W podrozdziale 2.2 zobaczymy, jak dzięki możliwości łączenia par można ich używać jako uniwersalnych elementów składowych do budowy wszelkiego rodzaju złożonych struktur danych. Jedna pierwotna złożona struktura danych — *para* zaimplementowana przez procedury `cons`, `car` i `cdr` — jest jedynym klejem, jakiego potrzebujemy. Obiekty danych zbudowane z par nazywamy *listowymi strukturami danych* (ang. *list-structured data*).

## Reprezentowanie liczb wymiernych

Pary stanowią naturalne uzupełnienie systemu liczb wymiernych. Liczbę wymierną reprezentujemy po prostu jako parę liczb całkowitych: licznik i mia-

<sup>2</sup> Nazwa `cons` oznacza „budować” (ang. *construct*). Nazwy `car` i `cdr` pochodzą z pierwszej implementacji Lispu na komputer IBM 704. Maszyna ta miała schemat adresowania umożliwiający odwoływanie się do dwóch części komórki pamięci: „adresu” i „zmniejszenia”. `Car` oznacza „zawartość adresowej części rejestru” (ang. „Contents of Address part of Register”), a `cdr` (czytaj: *kuder*) oznacza „zawartość zmniejszeniowej części rejestru” (ang. „Contents of Decrement part of Register”).

nownik. Wówczas `make-rat`, `numer` i `denom` można łatwo zaimplementować w następujący sposób<sup>3</sup>:

```
(define (make-rat n d) (cons n d))

(define (numer x) (car x))

(define (denom x) (cdr x))
```

W celu wyświetlenia wyników obliczeń możemy przedstawić liczbę wymierną, wypisując jej licznik, ukośnik i mianownik<sup>4</sup>:

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

Wypróbujmy teraz nasze procedury operujące na liczbach wymiernych:

```
(define one-half (make-rat 1 2))

(print-rat one-half)
1/2

(define one-third (make-rat 1 3))

(print-rat (add-rat one-half one-third))
5/6
```

---

<sup>3</sup> Inny sposób zdefiniowania tych selektorów i konstruktora to:

```
(define make-rat cons)
(define numer car)
(define denom cdr)
```

Pierwsza definicja kojarzy nazwę `make-rat` z wartością wyrażenia `cons`, którą jest procedura pierwotna budująca pary. Tak więc `make-rat` i `cons` stają się dwiema nazwami tego samego konstruktora pierwotnego.

Takie definiowanie selektorów i konstruktorów jest efektywne — `make-rat` zamiast *wywołać* procedurę `cons`, *jest* tą procedurą; zatem gdy wywołujemy `make-rat`, wywoływana jest tylko jedna procedura, a nie dwie. Jednakże zmniejsza to możliwości wspomagania odpluskowania przez śledzenie wywołań procedur lub ustawianie punktów kontrolnych w wywołaniach procedur — możemy chcieć śledzić wywołania `make-rat`, ale na pewno nie będziemy chcieli śledzić wszystkich wywołań `cons`.

Zdecydowaliśmy, że w niniejszej książce nie będziemy używać takiego stylu definiowania.

<sup>4</sup> `Display` to w języku Scheme procedura pierwotna służąca do wypisywania danych. Z kolei procedura pierwotna `newline` rozpoczęta wypisywanie od nowego wiersza. Żadna z tych procedur nie daje w wyniku żadnej użytecznej wartości, więc opisując zastosowania `print-rat`, podajemy tylko to, co wypisuje `print-rat`, a nie podajemy tego, co wypisuje interpreter jako wartość `print-rat`.

```
(print-rat (mul-rat one-half one-third))  
1/6  
  
(print-rat (add-rat one-third one-third))  
6/9
```

Jak pokazuje ostatni test, nasza implementacja liczb wymiernych nie skraca ułamków reprezentujących liczby wymierne. Możemy temu zaradzić, zmieniając procedurę `make-rat`. Gdybyśmy mieli procedurę `gcd`, taką jak ta przedstawiona w punkcie 1.2.5, której wynikiem jest największy wspólny dzielnik dwóch liczb całkowitych, to moglibyśmy jej użyć do skrócenia licznika i mianownika przed zbudowaniem pary:

```
(define (make-rat n d)  
  (let ((g (gcd n d)))  
    (cons (/ n g) (/ d g))))
```

Teraz otrzymujemy pożądany wynik:

```
(print-rat (add-rat one-third one-third))  
2/3
```

Dokonaliśmy tej modyfikacji, zmieniając konstruktor `make-rat` i nie zmieniając żadnej z procedur (takich jak `add-rat` i `mul-rat`) implementujących faktyczne operacje.

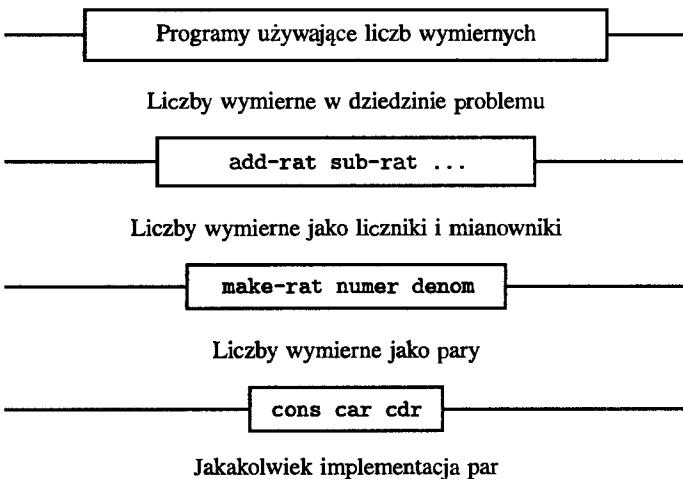
### Ćwiczenie 2.1

Zdefiniuj lepszą wersję `make-rat`, radzącą sobie z dodatnimi i ujemnymi argumentami. Procedura ta powinna normalizować znak, tzn. jeśli liczba wymierna jest dodatnia, to licznik i mianownik powinny być dodatnie, a jeśli jest ujemna, to tylko licznik powinien być ujemny.

#### 2.1.2. Bariery abstrakcji

Zanim przedstawimy dalsze przykłady danych złożonych i abstrakcji danych, rozważmy pewne kwestie powstałe przy okazji omawiania przykładu dotyczącego liczb wymiernych. Operacje na liczbach wymiernych zdefiniowaliśmy za pomocą konstruktora `make-rat` oraz selektorów `numer` i `denom`. Ogólnie mówiąc, zasadnicza koncepcja abstrakcji danych polega na zidentyfikowaniu dla każdego rodzaju obiektów danych podstawowego zestawu operacji, za pomocą których zostaną wyrażone wszystkie inne operacje na obiektach danych tego rodzaju, i operowaniu na tych danych tylko za pomocą tego zestawu operacji.

Możemy przewidzieć, że struktura systemu liczb wymiernych będzie taka, jak widać na rys. 2.1. Poziome linie przedstawiają *bariery abstrakcji* oddzielające różne „poziomy” systemu. Na każdym poziomie bariera oddziela



Rys. 2.1. Bariery abstrakcji w pakiecie liczb wymiernych

programy (te powyżej bariery) używające abstrakcji danych od programów (tych poniżej bariery) implementujących tę abstrakcję. Programy korzystające z liczb wymiernych operują na nich wyłącznie za pomocą procedur: `add-rat`, `sub-rat`, `mul-rat`, `div-rat` i `equal-rat?`, udostępnionych przez pakiet liczb wymiernych „do użytku publicznego”. Te procedury są z kolei zaimplementowane wyłącznie za pomocą konstruktora `make-rat` oraz selektorów `numer` i `denom`, które z kolei są zaimplementowane za pomocą par. Dopóki można operować na parach za pomocą `cons`, `car` i `cdr`, dopóty szczegóły ich implementacji są nieistotne dla reszty pakietu liczb wymiernych. W rezultacie procedury z każdego poziomu stanowią interfejsy łączące różne poziomy i określające bariery abstrakcji.

Ta prosta koncepcja ma wiele zalet. Jedną z nich jest to, że dużo łatwiej jest pielęgnować i modyfikować programy. Każda złożona struktura danych może być na wiele sposobów reprezentowana za pomocą pierwotnych struktur danych dostępnych w języku. Oczywiście wybór reprezentacji ma wpływ na korzystające z niej programy. Tak więc, jeśli kiedyś reprezentacja zostanie zmieniona, to wszystkie te programy mogą wymagać wprowadzenia odpowiednich modyfikacji. W przypadku dużych programów zadanie to może być czasochłonne i kosztowne — chyba że uzależnienie od sposobu reprezentacji danych zostanie ograniczone w projekcie programu do bardzo niewielu modułów.

Alternatywne podejście do problemu skracania ułamków reprezentujących liczby wymierne może na przykład polegać na skracaniu ich zawsze wtedy, gdy wydobywamy składowe ułamka, a nie wtedy, gdy go budujemy. Podejście takie prowadzi do innego konstruktora i selektorów:

```
(define (make-rat n d)
  (cons n d))

(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))

(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

Różnica między tą implementacją a poprzednią polega na tym, w którym momencie obliczamy gcd. Jeśli w typowym zastosowaniu liczb wymiernych wielokrotnie wydobywamy liczniki i mianowniki tych samych liczb wymiernych, to lepiej by było obliczać gcd, gdy budujemy liczby wymierne. Jeśli nie, to może nam się opłacić poczekać z obliczaniem gcd, dopóki nie będziemy wydobywać składowych ułamków. W każdym razie, gdy zmieniamy reprezentację z jednej na drugą, wówczas procedury `add-rat`, `sub-rat` itd. w ogóle nie muszą być zmieniane.

Ograniczanie uzależnienia od reprezentacji do kilku procedur interfejsu pomaga nam zarówno w konstruowaniu programów, jak i w modyfikowaniu ich, gdyż pozwala na zachowanie elastyczności i rozpatrywanie alternatywnych implementacji. Kontynuując nasz prosty przykład, założymy, że konstruując pakiet liczb wymiernych, nie możemy początkowo się zdecydować, czy wywoływać gcd w czasie budowania liczb wymiernych, czy też w czasie wydobywania ich składowych. Technika abstrakcji danych umożliwia nam odroczenie tej decyzji bez utraty możliwości rozwijania reszty systemu.

### Ćwiczenie 2.2

Rozważmy problem reprezentowania odcinków na płaszczyźnie. Każdy odcinek jest reprezentowany jako para punktów: jego „początek” i „koniec”. Zdefiniuj konstruktor `make-segment` oraz selektory `start-segment` i `end-segment`, definiujące reprezentację odcinka za pomocą punktów. Z kolei punkt można reprezentować jako parę liczb — współrzędnych  $x$  i  $y$ . W związku z tym zdefiniuj konstruktor `make-point` oraz selektory `x-point` i `y-point`, określające reprezentację punktów. Na koniec, korzystając ze swoich selektorów i konstruktorów, zdefiniuj procedurę `midpoint-segment`, której argumentem jest odcinek, a wynikiem jest jego środek (punkt, którego współrzędne są średnimi współrzędnych końców odcinka). Żeby przetestować swoje procedury, będzie Ci potrzebna procedura wypisująca punkty:

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

### Ćwiczenie 2.3

Zaimplementuj reprezentację prostokątów na płaszczyźnie [o bokach równoległych do osi układu współrzędnych; przyp. tłum.]. (Wskazówka: Mogą Ci się przydać wyniki ćwiczenia 2.2). Korzystając ze swoich konstruktorów i selektorów, napisz procedury obliczające obwód i powierzchnię danego prostokąta. Następnie zaimplementuj inną reprezentację prostokątów. Czy potrafisz tak skonstruować swój system, stosując odpowiednie bariery abstrakcji, aby te same procedury obliczające obwód i powierzchnię działały przy obydwu reprezentacjach?

#### 2.1.3. Co rozumiemy przez dane?

W punkcie 2.1.1 rozpoczęliśmy implementację liczb wymiernych od zaimplementowania operacji na liczbach wymiernych (`add-rat`, `sub-rat` itd.) przy użyciu trzech nieokreślonych procedur: `make-rat`, `numer` i `denom`. W tym momencie moglibyśmy uważać, że operacje te są zdefiniowane za pomocą obiektów danych — liczników, mianowników i liczb wymiernych — których zachowanie jest określone przez trzy pozostałe procedury.

Co jednak dokładnie rozumiemy przez *dane*? Nie wystarczy powiedzieć „cokolwiek, co jest zaimplementowane przez konstruktory i selektory”. Oczywiście, nie każde dowolne trzy procedury mogą służyć jako odpowiednia podstawa implementacji liczb wymiernych. Musimy zapewnić, że jeżeli z pary liczb całkowitych  $n$  i  $d$  zbudujemy liczbę wymierną  $x$ , a następnie podzielimy przez siebie wyniki procedur `numer` i `denom` zastosowanych do  $x$ , to uzyskamy ten sam wynik co po podzieleniu  $n$  przez  $d$ . Innymi słowy, `make-rat`, `numer` i `denom` muszą spełniać warunek, że dla dowolnej liczby całkowitej  $n$  i dla dowolnej liczby całkowitej  $d$  różnej od zera zachodzi: jeśli  $x$  jest wynikiem `(make-rat n d)`, to

$$\frac{(\text{numer } x)}{(\text{denom } x)} = \frac{n}{d}$$

W rzeczywistości jest to jedyny warunek, jaki muszą spełniać `make-rat`, `numer` i `denom`, aby tworzyć odpowiednią reprezentację liczb wymiernych. Ogólnie mówiąc, możemy myśleć o danych jako o czymś zdefiniowanym przez pewien zestaw selektorów i konstruktorów wraz z określonymi warunkami, które procedury te muszą spełniać, aby być poprawną reprezentacją danych<sup>5</sup>.

<sup>5</sup> Jest to zaskakujące, ale bardzo trudno jest ściśle sformułować to pojęcie. Są dwie metody formułowania go. Jedna z nich, której pionierem jest C. A. R. Hoare [51], jest znana jako metoda *modeli abstrakcyjnych* (ang. *abstract models*). Formalizuje ona specyfikację „procedur wraz z warunkami”, jak było to przedstawione w zarysie w powyższym przykładzie liczb wymiernych. Zauważmy, że warunek opisujący reprezentację liczb wymiernych był wyrażony za pomocą faktów dotyczących liczb całkowitych (równość i dzielenie). Ogólnie mówiąc, modele abstrakcyjne definiują nowe rodzaje obiektów danych za pomocą uprzednio zdefiniowanych rodzajów obiektów danych. Stwierdzenia dotyczące obiektów danych mogą więc być

Ten punkt widzenia dotyczy nie tylko obiektów danych „wysokiego poziomu”, takich jak liczby wymierne, ale także obiektów danych niskiego poziomu. Rozważmy pojęcie pary, z którego korzystaliśmy przy definiowaniu liczb wymiernych. Nigdy, faktycznie, nie powiedzieliśmy, czym jest para; stwierdziliśmy tylko, że język programowania dostarcza procedur `cons`, `car` i `cdr`, służących do operowania na parach. Jednak jedyną rzeczą, którą musimy wiezieć o tych trzech operacjach, jest to, że jeżeli sklejmy dwa obiekty razem za pomocą `cons`, to możemy odzyskać obiekty składowe za pomocą `car` i `cdr`. Oznacza to, że operacje te spełniają taki warunek, że: dla dowolnych obiektów `x` i `y`, jeśli `z` jest wynikiem `(cons x y)`, to `(car z)` jest obiektem `x`, a `(cdr z)` jest obiektem `y`. Istotnie, wspomnieliśmy, że te trzy procedury są elementami pierwotnymi naszego języka. Jednakże dowolne trzy procedury spełniające powyższy warunek mogą posłużyć jako podstawa do zaimplementowania par. Kwestię tę uderzająco ilustruje fakt, że moglibyśmy zaimplementować `cons`, `car` i `cdr`, nie używając żadnych struktur danych, a korzystając jedynie z procedur. Oto definicje:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument różny od 0 i 1 -- CONS" m))))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

Takie użycie procedur nie odpowiada wcale naszemu intuicyjnemu wyobrażeniu, czym powinny być dane. Niemniej jednak, aby wykazać, że jest to poprawna reprezentacja par, wystarczy jedynie sprawdzić, że procedury te spełniają powyższy warunek.

Zauważmy subtelny fakt, że wynik `(cons x y)` jest procedurą — mianowicie jednoargumentową procedurą lokalną `dispatch`, która daje w wyniku `x` albo `y` w zależności od tego, czy jej argumentem jest 0 czy 1. Zgodnie z tym `(car z)` jest zdefiniowane jako zastosowanie `z` do 0. Stąd, jeśli `z` jest procedurą powstałą w wyniku `(cons x y)`, to wynikiem zastosowania `z` do

---

sprawdzane poprzez sprowadzenie ich do stwierdzeń dotyczących uprzednio zdefiniowanych obiektów danych. Druga metoda, zaproponowana przez Zillesa z MIT, Goguena, Thatcher, Wagnera i Wrighta z IBM [104] oraz przez Guttaga z Toronto [39], jest nazywana *specyfikacją algebraiczną* (ang. *algebraic specification*). Patrzymy tu na „procedury” jak na elementy abstrakcyjnych systemów algebraicznych, których zachowanie jest opisywane za pomocą aksjomatów odpowiadających naszym „warunkom”. Sprawdzając stwierdzenia dotyczące danych, korzystamy z technik algebrai abstrakcyjnej. Przegląd obydwu metod można znaleźć w pracy Liskov i Zillesa [68].

0 jest x. Pokazaliśmy więc, że wynikiem (`car (cons x y)`) jest, tak jak chcieliśmy, x. Podobnie, (`cdr (cons x y)`) stosuje procedurę będącą wynikiem (`cons x y`) do 1, czego wynikiem jest y. Tak więc ta proceduralna implementacja par jest poprawna i jeśli będziemy korzystać z par jedynie za pomocą `cons`, `car` i `cdr`, to nie będziemy w stanie jej odróżnić od „prawdziwej” struktury danych.

Istotą proceduralnej reprezentacji par nie jest to, że nasz język działa w ten sposób (Scheme i w ogóle systemy lispowe ze względów wydajnościowych implementują pary wprost), ale to, że mógłby tak działać. Reprezentacja proceduralna, choć niejasna, jest całkowicie odpowiednim sposobem reprezentowania par, gdyż spełnia jedyny warunek, jaki pary muszą spełniać. Przykład ten pokazuje też, że możliwość traktowania procedur jako obiektów automatycznie daje nam możliwość reprezentowania danych złożonych. Teraz może się nam to wydawać osobliwe, ale proceduralne reprezentacje danych będą odgrywały główną rolę w naszym programistycznym repertuarze. Taki styl programowania jest często nazywany *przekazywaniem komunikatów* (ang. *message passing*) i będzie on naszym podstawowym narzędziem w rozdziale 3, gdzie zajmiemy się problemami modelowania i symulacji.

### Ćwiczenie 2.4

Oto alternatywna proceduralna reprezentacja par. Sprawdź, czy przy tej reprezentacji dla dowolnych obiektów x i y wynikiem (`car (cons x y)`) jest x.

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))
```

Jaka powinna być odpowiednia definicja `cdr`? (Wskazówka: Aby sprawdzić, że to działa, skorzystaj z modelu podstawieniowego opisanego w punkcie 1.1.5).

### Ćwiczenie 2.5

Pokaż, że można reprezentować pary nieujemnych liczb całkowitych, używając tylko liczb i operacji arytmetycznych, jeśli reprezentujemy parę  $a$  i  $b$  jako iloczyn  $2^a 3^b$ . Podaj odpowiednie definicje procedur `cons`, `car` i `cdr`.

### Ćwiczenie 2.6

Na wypadek, gdyby proceduralna reprezentacja par nie zamieszała Ci dostatecznie w głowie, rozważ sytuację, gdy w języku, w którym mamy możliwość operowania na procedurach, możemy obyć się bez liczb (przynajmniej bez nieujemnych liczb całkowitych), implementując 0 i operację zwiększania o 1 jako:

```
(define zero (lambda (f) (lambda (x) x)))

(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

Taka reprezentacja jest znana jako *liczby Churcha*, od nazwiska ich twórcy — logika Alonzo Churcha, który wymyślił rachunek  $\lambda$ .

Zdefiniuj bezpośrednio (tzn. bez użycia zero i add-1) stałe jeden i dwa. (Wskaźówka: Użyj modelu podstawieniowego do obliczenia wartości (add-1 zero)). Podaj bezpośrednią definicję procedury dodawania + (ale nie za pomocą wielokrotnego stosowania add-1).

#### 2.1.4. Rozszerzone ćwiczenie: arytmetyka przedziałów

Liz P. Haker konstruuje system pomagający w rozwiązywaniu problemów technicznych. Jedną z cech, jakie chce, aby miał jej system, jest możliwość operowania na wartościach przybliżonych (takich jak wartości mierzone przez urządzenia fizyczne) mierzonych z określoną precyzją, tak aby po zakończeniu obliczeń na takich przybliżonych wartościach była znana dokładność wyników.

Inżynierowie elektrycy mogą używać systemu Liz do obliczania wielkości elektrycznych. Czasami potrzebują oni obliczyć rezystancję wypadkową  $R_p$  w przypadku równoległego połączenia dwóch rezystorów (obwodu równoległego)  $R_1$  i  $R_2$ , korzystając ze wzoru

$$R_p = \frac{1}{1/R_1 + 1/R_2}$$

Wartości rezystancji są zazwyczaj znane tylko z pewną tolerancją gwarantowaną przez producenta rezystorów. Jeśli na przykład kupimy rezystor oznaczony „ $6.8 \Omega \pm 10\%$ ”, możemy być jedynie pewni, że jego rezystancja należy do przedziału od  $6.8 - 0.68 = 6.12 \Omega$  do  $6.8 + 0.68 = 7.48 \Omega$ . Tak więc, jeśli połączymy równolegle rezystory o wartościach rezystancji  $6.8 \Omega \pm 10\%$  i  $4.7 \Omega \pm 5\%$ , to rezystancja wypadkowa może się ważyć od około  $2.58 \Omega$  (gdy obydwa rezystory mają minimalne dopuszczalne rezystancje) do około  $2.97 \Omega$  (gdy obydwa rezystory mają maksymalne dopuszczalne rezystancje).

Pomysł Liz polega na zaimplementowaniu „arytmetyki przedziałów” — zestawu operacji arytmetycznych na „przedziałach” (obiektach reprezentujących zakresy możliwych wartości wielkości przybliżonych). Wynik dodawania, odejmowania, mnożenia bądź dzielenia dwóch przedziałów sam jest przedziałem reprezentującym zakres możliwych wyników.

Liz postuluje wprowadzenie abstrakcyjnego obiektu nazywanego „przedziałem”, który ma dwa punkty końcowe oznaczone jako granica dolna (ang. *lower bound*) i granica górna (ang. *upper bound*). Przyjmuje ona również, że znając punkty końcowe przedziału, można zbudować ten przedział za pomocą konstruktora `make-interval`. Liz pisze najpierw procedurę dodawania dwóch przedziałów. Rozumie ona następująco: najmniejsza możliwa wartość sumy jest sumą dwóch granic dolnych, a największa możliwa wartość sumy jest sumą dwóch granic górnych:

---

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                (+ (upper-bound x) (upper-bound y))))
```

Liz opracowuje również mnożenie dwóch przedziałów, wyznaczając minimum i maksimum z iloczynów granic przedziałów i używając ich jako granic wynikowego przedziału. (Min i max to procedury pierwotne znajdujące minimum i maksimum dla dowolnej liczby argumentów).

```
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y))))
    (p2 (* (lower-bound x) (upper-bound y))))
    (p3 (* (upper-bound x) (lower-bound y))))
    (p4 (* (upper-bound x) (upper-bound y))))
  (make-interval (min p1 p2 p3 p4)
                (max p1 p2 p3 p4))))
```

Aby podzielić przez siebie dwa przedziały, Liz mnoży pierwszy z nich przez odwrotność drugiego. Zauważmy, że dolna i górna granica odwrotności przedziału to, odpowiednio, odwrotność górnej granicy i odwrotność dolnej granicy tego przedziału\*.

```
(define (div-interval x y)
  (mul-interval x
    (make-interval (/ 1.0 (upper-bound y))
                  (/ 1.0 (lower-bound y))))))
```

### Ćwiczenie 2.7

Program Liz jest niedokończony, ponieważ nie określiła ona implementacji abstrakcji przedziałów. Oto definicja konstruktora przedziałów:

```
(define (make-interval a b) (cons a b))
```

Dokończ implementację, definiując selektory `upper-bound` i `lower-bound`.

### Ćwiczenie 2.8

Rozumując podobnie jak Liz, opisz, jak można obliczyć różnicę dwóch przedziałów. Zdefiniuj odpowiednią procedurę odejmowania o nazwie `sub-interval`.

### Ćwiczenie 2.9

*Szerokość* (ang. *width*) przedziału to połowa różnicy między jego górną i dolną granicą. Szerokość jest miarą niepewności co do wartości liczby określonej przez przedział. W przypadku niektórych operacji arytmetycznych szerokość przedziału powstałego w wyniku wykonania operacji na dwóch przedziałach jest funkcją szerokości tych

---

\* Jest to prawdą tylko dla przedziałów nie zawierających zera (przyp. tłum.).

przedziałów, podczas gdy w przypadku innych operacji szerokości przedziału wynikowej nie da się określić jako funkcji szerokości argumentów operacji. Pokaż, że szerokość sumy (lub różnicy) dwóch przedziałów jest funkcją zależną tylko od szerokości dodawanych (lub odejmowanych) przedziałów. Podaj przykłady świadczące o tym, że w przypadku mnożenia i dzielenia tak nie jest.

### Ćwiczenie 2.10

Ben Bajerbit, wytrawny programista systemów, zajrzał Liz przez ramię i zauważył, że nie jest jasne, co oznacza dzielenie przez przedział zawierający zero. Popraw tak program Liz, aby sprawdzał, czy zachodzi taka sytuacja, i zgłaszał błąd, jeśli tak jest.

### Ćwiczenie 2.11

Przy okazji Ben stwierdził również zagadkowo, że: „Sprawdzając znaki granic przedziałów, można rozbić mul-interval na dziewięć przypadków, z czego tylko w jednym potrzeba więcej niż dwóch mnożeń”. Napisz jeszcze raz tę procedurę zgodnie z sugestią Bena.

Po usunięciu błędów ze swojego programu Liz pokazuje go potencjalnemu użytkownikowi, który narzeka, że jej program rozwiązuje niewłaściwy problem. Potrzebny mu jest program, który potrafi operować na liczbach reprezentowanych jako środek przedziału plus pewna tolerancja; wolałby na przykład działać raczej na liczbach postaci  $3.5 \pm 0.15$  niż postaci  $[3.35, 3.65]$ . Liz wraca do biurka i rozwiązuje ten problem, opracowując alternatywny konstruktor i selektory:

```
(define (make-center-width c w)
  (make-interval (- c w) (+ c w)))

(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))

(define (width i)
  (/ (- (upper-bound i) (lower-bound i)) 2))
```

Niestety, większość użytkowników programu Liz to inżynierowie. W prawdziwych zastosowaniach inżynierskich pomiary są zwykle obarczone tylko niewielkim błędem, mierzonym jako stosunek szerokości przedziału do jego wartości środkowej. Inżynierowie zwykle określają tolerancję parametrów urządzeń w procentach, jak widzieliśmy to na przykładzie specyfikacji rezystora.

### Ćwiczenie 2.12

Zdefiniuj konstruktor `make-center-percent` budujący przedział na podstawie jego środka i tolerancji określonej w procentach. Musisz również zdefiniować selektor `percent`, którego wynikiem jest podana w procentach tolerancja danego przedziału. Selektor `center` pozostaje taki sam jak powyżej.

### Ćwiczenie 2.13

Pokaż, przy założeniu, że tolerancja jest niewielkim procentem, iż istnieje prosty wzór przybliżający procentową tolerancję iloczynu dwóch przedziałów za pomocą tolerancji czynników. Dla uproszczenia możesz przyjąć, że wszystkie liczby są dodatnie.

Liz P. Haker, wykonawszy kawał roboty, przekazała ukończony system. Kilka lat później, gdy o wszystkim już zapomniała, odebrała szalony telefon od rozgniewanego użytkownika, Jana A. Prawie. Zdaje się, że Jan zauważył, iż wzór na rezystancję wypadkową równolegle połączonych rezystorów może być zapisany na dwa algebraicznie równoważne sposoby:

$$\frac{R_1 R_2}{R_1 + R_2}$$

lub

$$\frac{1}{1/R_1 + 1/R_2}$$

Napisał on dwa następujące programy, z których każdy oblicza tę rezystancję w inny sposób:

```
(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval one
                  (add-interval (div-interval one r1)
                                (div-interval one r2)))))
```

Jan narzeka, że program Liz daje różne wyniki przy dwóch różnych metodach obliczania. Jest to poważna reklamacja.

### Ćwiczenie 2.14

Pokaż, że Jan ma rację. Zbadaj zachowanie systemu dla różnych wyrażeń arytmetycznych. Zdefiniuj pewne przedziały  $A$  i  $B$  i użyj ich do obliczenia wyrażeń  $A/A$  i  $A/B$ . Najlepszy wgląd dadzą Ci przedziały, których szerokość jest niewielkim procentem ich wartości środkowej. Zbadaj wyniki obliczeń przedstawione w postaci wartości środkowej i procentowej tolerancji (zob. ćwiczenie 2.12).

### Ćwiczenie 2.15

Inny użytkownik, Ewa Lu Ator, także zauważyła, że różne, ale algebraicznie równoważne wyrażenia dają w wyniku różne przedziały. Twierdzi ona, że obliczanie na przedziałach, za pomocą systemu Liz, wartości wzoru da lepsze oszacowanie błędu wtedy, gdy wzór będzie zapisany w takiej postaci, że żadna zmienna reprezentująca

przybliżoną wartość nie będzie się powtarzać. Tak więc, jak twierdzi, `par2` jest „lepszym” programem obliczającym rezystancję obwodu równoległego niż `par1`. Czy ma rację? Dlaczego?

### Ćwiczenie 2.16

Wyjaśnij ogólnie, dlaczego algebraicznie równoważne wyrażenia mogą dawać różne wyniki. Czy potrafisz wymyślić pakiet arytmetyki przedziałów pozbawiony tej nie-dogodności, czy też jest to niemożliwe? (Uwaga: Jest to bardzo trudny problem).

## 2.2. Dane hierarchiczne i własność domknięcia

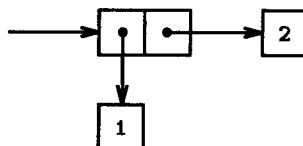
Jak widzieliśmy, pary stanowią rodzaj pierwotnego „kleju”, którego możemy użyć do budowy złożonych obiektów danych. Na rysunku 2.2 widać standardowy sposób graficznego przedstawiania pary — tutaj pary powstałej w wyniku `(cons 1 2)`. W reprezentacji tej, nazywanej *pudełkowo-wskaźnikową* (ang. *box-and-pointer*), każdy obiekt jest ukazany jako *wskaźnik* do pudełka. W przypadku obiektu pierwotnego pudełko zawiera jego reprezentację. Na przykład pudełko liczby zawiera jej wartość. Pudełko pary jest w rzeczywistości po-dwójnym pudełkiem, którego lewa połowa zawiera (*wskaźnik do*) `car` pary, a prawa połowa zawiera (*wskaźnik do*) `cdr`.

Widzieliśmy już, że za pomocą `cons` można łączyć nie tylko liczby, ale również pary. (Z faktu tego trzeba było skorzystać w ćwiczeniach 2.2 i 2.3). Tak więc pary są uniwersalnymi elementami składowymi, z których możemy budować wszystkie rodzaje struktur danych. Na rysunku 2.3 widać dwa sposoby połączenia liczb 1, 2, 3 i 4 za pomocą `par`.

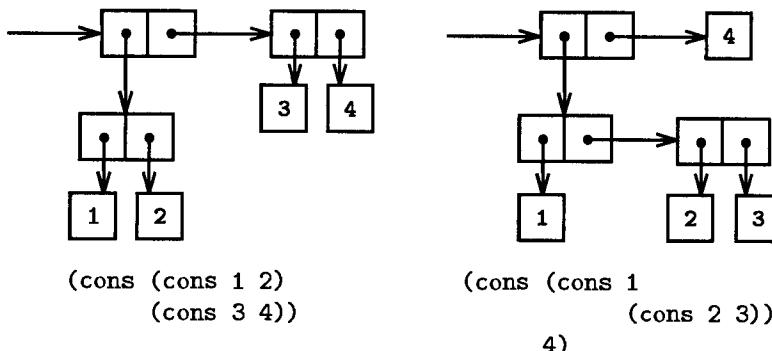
Możliwość tworzenia par, których elementami są pary, stanowi istotę struktury listowej jako sposobu reprezentacji. Możliwość tę nazywamy *własnością domknięcia* `cons`. Ogólnie mówiąc, operacja służąca do łączenia obiektów danych ma własność domknięcia, jeśli wyniki łączenia za pomocą tej operacji mogą same być łączone za pomocą tej operacji<sup>6</sup>. Własność domknięcia ma podstawowe znaczenie w przypadku każdego sposobu łączenia danych, gdyż pozwala ona na tworzenie struktur *hierarchicznych* — struktur złożonych z części, które same są złożone z części itd.

Od samego początku rozdziału 1 w istotny sposób korzystaliśmy z własności domknięcia procedur, gdyż wszystkie nasze programy, z wyjątkiem najprostszym, opierały się na tym, że elementy kombinacji mogą same być kombinacjami. W niniejszym podrozdziale zajmiemy się konsekwencjami własności

<sup>6</sup> Użyte tutaj słowo „domknięcie” pochodzi z algebry abstrakcyjnej, gdzie o zbiorze elementów mówi się, że jest domknięty ze względu na daną operację, jeśli stosowanie tej operacji do elementów zbioru daje w wyniku elementy należące do tego samego zbioru. Społeczność lispowa używa (niestety) również słowa „domknięcie” w odniesieniu do zupełnie innego pojęcia — domknięcie jest techniką implementacyjną służącą do reprezentowania procedur ze zmiennymi wolnymi. W tej książce nie używamy słowa „domknięcie” w tym drugim znaczeniu.



Rys. 2.2. Pudełkowo-wskaźnikowa reprezentacja (cons 1 2)



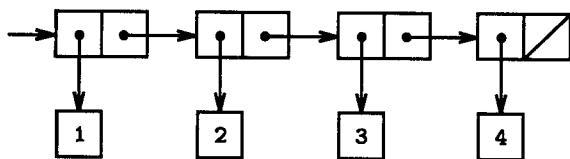
Rys. 2.3. Dwa sposoby połączenia liczb 1, 2, 3 i 4 za pomocą par

domknięcia danych złożonych. Opiszemy pewne typowe metody reprezentowania ciągów i drzew za pomocą par oraz przedstawimy język graficzny wyraziście ilustrujący własność domknięcia<sup>7</sup>.

### 2.2.1. Reprezentowanie ciągów

Jedną z użytecznych struktur, jaką możemy zbudować za pomocą par, jest *ciąg* (ang. *sequence*) — uporządkowana kolekcja obiektów danych. Istnieje oczywiście wiele sposobów reprezentowania ciągów za pomocą par. Jedna szczególnie prosta reprezentacja jest pokazana na rys. 2.4, gdzie ciąg (1, 2, 3, 4) przedstawiono jako łańcuch par. Dla każdej pary car stanowi odpowiadający jej element ciągu, a cdr jest kolejną parą w łańcuchu. Cdr dla ostatniej

<sup>7</sup> Pogląd, że środki łączenia powinny mieć własność domknięcia jest prostą koncepcją. Niestety, środki łączenia danych dostępne w wielu popularnych językach programowania nie spełniają własności domknięcia lub czynią ją nieporęczną w użyciu. W Fortranie lub Basicu zwykle łączy się elementy danych w tablicę — jednak nie można utworzyć tablic, których elementami byłyby tablice. Pascal i C pozwalają na tworzenie struktur danych, których elementami są struktury danych. Wymaga to jednak od programisty jawnego operowania na wskaźnikach i narzuca ograniczenie polegające na tym, że struktury danych mogą zawierać tylko elementy uprzednio określonej postaci. Języki te, w odróżnieniu od Lispu, nie mają żadnego wbudowanego uniwersalnego kleju ułatwiającego operowanie na złożonych danych w jednolity sposób. Ograniczenia te kryją się za komentarzem Alana Perlisa zawartym w słowie wstępny do niniejszej książki: W Pascalu mnóstwo deklarowalnych struktur danych powoduje specjalizację funkcji, która hamuje i utrudnia ich doraźne współdziałanie. Lepiej mieć 100 funkcji, które działają na jednej strukturze danych, niż 10, które działają na 10 strukturach danych.



Rys. 2.4. Ciąg (1, 2, 3, 4) reprezentowany jako łańcuch par

para sygnalizuje koniec ciągu, wskazując na wyróżnioną wartość nie będącą parą, reprezentowaną na diagramach pudełkowo-wskaźnikowych przez przekątną, a w programach przez wartość zmiennej `nil`. Cały ciąg konstruujemy za pomocą zagnieżdżonych operacji `cons`:

```
(cons 1
  (cons 2
    (cons 3
      (cons 4 nil))))
```

Taki ciąg par, utworzony przez zagnieżdżone operacje `cons`, jest nazywany *listą* (ang. *list*). Schemat udostępnia procedurę pierwotną `list` pomagającą w budowaniu listy<sup>8</sup>. Powyższy ciąg mógłby być wynikiem `(list 1 2 3 4)`. Ogólnie mówiąc

```
(list <a1> <a2> ... <an>)
```

jest równoważne

```
(cons <a1> (cons <a2> (cons ... (cons <an> nil) ...)))
```

Systemy lispowe tradycyjnie wypisują listy jako ciąg elementów ujętych w nawiasy okrągłe. Tak więc obiekt danych przedstawiony na rys. 2.4 zostałby wypisany jako `(1 2 3 4)`:

```
(define one-through-four (list 1 2 3 4))
```

```
one-through-four
(1 2 3 4)
```

Należy być ostrożnym i nie mylić wyrażenia `(list 1 2 3 4)` z listą `(1 2 3 4)`, która jest wynikiem obliczenia wartości tego wyrażenia. Próba obliczenia wartości wyrażenia `(1 2 3 4)` spowoduje błąd, kiedy interpreter spróbuje zastosować procedurę `1` do argumentów `2, 3 i 4`.

<sup>8</sup> W niniejszej książce używamy terminu *lista* na określenie łańcucha par zakończonego znakiem końca listy. Z kolei termin *struktura listowa* (ang. *list structure*) odnosi się do dowolnej struktury danych zbudowanej za pomocą par, a nie tylko do list.

Możemy patrzeć na `car` jak na operację, której wynikiem jest pierwszy element listy, a na `car` jak na operację, której wynikiem jest podlista zawierającą wszystkie elementy oprócz pierwszego. Za pomocą zagnieżdżonych wywołań `car` i `cdr` można odwoływać się do drugiego, trzeciego i kolejnych elementów listy<sup>9</sup>. Konstruktor `cons` tworzy listę powstałą przez dodanie dodatkowego elementu na początku danej listy.

```
(car one-through-four)
1

(cdr one-through-four)
(2 3 4)

(car (cdr one-through-four))
2

(cons 10 one-through-four)
(10 1 2 3 4)

(cons 5 one-through-four)
(5 1 2 3 4)
```

Wartość `nil`, używaną do oznaczania końca łańcucha par, można traktować jako *listę pustą* (ang. *empty list*) — nie zawierającą żadnych elementów. Słowo `nil` jest skróconą formą łacińskiego słowa *nihil*, oznaczającego „nic”<sup>10</sup>.

### Operacje na listach

Zastosowaniu par do reprezentacji ciągów elementów w postaci list towarzyszą typowe techniki programistyczne przetwarzania list polegające na „cdrowaniu wzduż” listy kolejnych jej elementów. Na przykład argumentami procedury `list-ref` są lista i liczba  $n$ , a wynikiem jest  $n$ -ty element danej listy. Zwycza-

<sup>9</sup> Ponieważ zagnieżdżone wywołania `car` i `cdr` są nieporęczne, dialekty Lispu udostępniają skróconą formę ich zapisu — na przykład:

```
(cadr <arg>) = (car (cdr <arg>))
```

Nazwy wszystkich takich procedur zaczynają się literą `c`, a kończą literą `r`. Każde a między nimi oznacza operację `car`, a każde `d` oznacza operację `cdr` — złożone w takiej samej kolejności, w jakiej pojawiają się w nazwie. Nazwy `car` i `cdr` utrzymano, gdyż ich proste kombinacje, jak `cadr`, dają się łatwo wymówić.

<sup>10</sup> Niezwykłe jest to, jak dużo energii zmarnowano w trakcie standaryzacji dialektów Lispu na spory dosłownie o „nic”. Czy `nil` powinno być zwykłą nazwą? Czy wartość `nil` powinna być symbolem? Czy powinna być listą? Czy powinna być parą? W języku Scheme `nil` jest znacznik końca listy (tak jak `true` jest zwykłą zmienną, której wartością jest prawda). Inne dialekty Lispu, w tym Common Lisp, traktują `nil` jako symbol specjalny. Autorzy niniejszej książki przeszli przez zbyt wiele kłótni standaryzacyjnych i woleliby uniknąć całej tej kwestii. Od momentu, gdy w podrozdziale 2.3 wprowadzimy cytowanie, będziemy oznaczać listę pustą przez `'()` i pozbędziemy się całkowicie zmiennej `nil`.

jowo elementy listy numeruje się od zera. Metoda obliczania list-ref jest następująca:

- Dla  $n = 0$  wynikiem list-ref powinien być car listy.
- W przeciwnym razie wynikiem list-ref powinien być  $(n-1)$ -szy element cdr listy.

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))  
  
(define squares (list 1 4 9 16 25))  
  
(list-ref squares 3)  
16
```

Będziemy często cdrowali wzdłuż całej listy. Scheme wspomaga takie po- stępowanie i udostępnia predykat pierwotny `null?`, który sprawdza, czy jego argument jest listą pustą. Procedura `length`, obliczająca liczbę elementów li- sty, przedstawia typowy sposób użycia tego predykatu:

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))  
  
(define odds (list 1 3 5 7))  
  
(length odds)  
4
```

Procedura `length` jest zaimplementowana według prostego schematu rekuren- cyjnego. Krok redukcyjny jest następujący:

- Długość dowolnej listy jest równa 1 plus długość cdr tej listy.

Stosując ten krok wielokrotnie, dochodzimy do przypadku podstawowego:

- Długość listy pustej wynosi 0.

Moglibyśmy również obliczać `length` iteracyjnie:

```
(define (length items)
  (define (length-iter a count)
    (if (null? a)
        count
```

```
(length-iter (cdr a) (+ 1 count)))
(length-iter items 0))
```

Inna typowa technika programowania polega na „konstruowaniu” listy wynikowej równocześnie z cdrowaniem wzduż danej listy, jak ma to miejsce w procedurze `append`, której wynikiem jest nowa lista powstała przez sklejenie dwóch list będących jej argumentami:

```
(append squares odds)
(1 4 9 16 25 1 3 5 7)
```

```
(append odds squares)
(1 3 5 7 1 4 9 16 25)
```

Procedura `append` jest zaimplementowana również według schematu rekurencyjnego. Chcąc skleić ze sobą listy `list1` i `list2`, należy wykonać, co następuje:

- Jeśli `list1` jest listą pustą, to wynikiem jest po prostu `list2`.
- W przeciwnym razie trzeba skleić `cdr` listy `list1` z listą `list2` i z tak powstałej listy oraz `car` listy `list1` należy skonstruować wynik:

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```

### Ćwiczenie 2.17

Zdefiniuj procedurę `last-pair`, której wynikiem jest lista zawierająca tylko ostatni element danej (niepustej) listy:

```
(last-pair (list 23 72 149 34))
(34)
```

### Ćwiczenie 2.18

Zdefiniuj procedurę `reverse`, której wynikiem jest lista powstała przez odwrócenie listy będącej jej argumentem:

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

### Ćwiczenie 2.19

Rozważmy problem wydawania reszty, opisany w punkcie 1.2.2. Byłoby miło, gdybyśmy mogli łatwo zmieniać walutę używaną przez program i móc na przykład obliczyć, na ile sposobów możemy wydać resztę równą jednemu brytyjskiemu funtowi. Program jest tak napisany, że informacje o walucie są rozdzielone między dwie procedury:

first-denomination i count-change (która wie, że w USA jest pięć rodzajów monet). Byłoby lepiej, gdybyśmy mogli podawać listę monet, jakich można używać do wydawania reszty.

Chcielibyśmy tak zmienić procedurę cc, aby jej drugi argument był listą nominałów używanych monet, a nie liczbą całkowitą określającą, których monet można użyć. Wówczas moglibyśmy mieć listy określające wszystkie waluty:

```
(define us-coins (list 50 25 10 5 1))  
  
(define uk-coins (list 100 50 20 10 5 2 1 0.5))
```

i wywoływać cc w następujący sposób:

```
(cc 100 us-coins)  
292
```

Wymagałoby to jednak pewnych zmian w programie cc. Miałby on taką samą postać, ale inaczej odwoływałby się do swojego drugiego argumentu; robiłby to następująco:

```
(define (cc amount coin-values)  
  (cond ((= amount 0) 1)  
        ((or (< amount 0) (no-more? coin-values)) 0)  
        (else  
          (+ (cc amount  
                  (except-first-denomination coin-values))  
              (cc (- amount  
                      (first-denomination coin-values))  
                  coin-values))))
```

Zdefiniuj procedury first-denomination, except-first-denomination i no-more? za pomocą operacji pierwotnych na strukturach listowych. Czy kolejność elementów na liście coin-values wpływa na wynik cc? Wyjaśnij, dlaczego tak lub dlaczego nie?

### Ćwiczenie 2.20

Procedury +, \* i list mogą mieć dowolną liczbę argumentów. Jeden ze sposobów definiowania takich procedur polega na zastosowaniu w define notacji kropki i ogona (ang. *dotted-tail notation*). W definicji procedury, na liście parametrów, przed ostatnim parametrem może znajdować się kropka — wskazuje ona, że w chwili wywołania procedury początkowe parametry (jeśli występują) przyjmują, jak zwykle, wartości początkowych argumentów, a wartością ostatniego parametru jest lista wartości pozostałych argumentów. Mając na przykład taką definicję:

```
(define (f x y . z) <treść>)
```

możemy wywoływać procedurę f z dwoma lub więcej argumentami. Jeśli wywołamy

```
(f 1 2 3 4 5 6)
```

to w treści procedury f: x będzie równe 1, y będzie równe 2, a z będzie listą (3 4 5 6). Przy definicji postaci

```
(define (g . w) <treść>)
```

możemy wywoływać procedurę g z dowolną liczbą argumentów. Jeśli wywołamy

```
(g 1 2 3 4 5 6)
```

to w treści procedury g: w będzie listą (1 2 3 4 5 6)<sup>11</sup>.

Użyj tej notacji do zapisania procedury same-parity mającej jeden lub więcej argumentów będących liczbami całkowitymi i dającej w wyniku listę złożoną z tych wszystkich argumentów, które są tak samo jak pierwszy argument parzyste lub nieparzyste. Na przykład:

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
```

```
(same-parity 2 3 4 5 6 7)
(2 4 6)
```

## Odwzorowywanie list

Jedna z niezmiernie użytecznych operacji polega na zastosowaniu danego przekształcenia do wszystkich elementów listy i utworzeniu listy wyników. Na przykład następująca procedura mnoży wszystkie liczby na liście przez zadany czynnik:

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
```

```
(scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

Możemy dokonać abstrakcji tej ogólnej koncepcji i wyrazić wspólny wzorzec takich procedur za pomocą procedury wyższego rzędu — tak jak to robiliśmy w podrozdziale 1.3. Tę procedurę wyższego rzędu nazwiemy map. Argumentami map są procedura jednoargumentowa i lista, a wynikiem jest lista

---

<sup>11</sup> Chcąc zdefiniować f i g za pomocą lambda-abstrakcji, musimy je zapisać jako

```
(define f (lambda (x y . z) <treść>))
(define g (lambda w <treść>))
```

utworzona z wyników zastosowania danej procedury do kolejnych elementów danej listy<sup>12</sup>:

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))

(map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)

(map (lambda (x) (* x x))
      (list 1 2 3 4))
(1 4 9 16)
```

Możemy teraz zdefiniować `scale-list` za pomocą `map`:

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))
```

Map jest ważną konstrukcją nie tylko dlatego, że ujmuje wspólny wzorzec procedur, ale dlatego, że ustala wyższy poziom abstrakcji w operowaniu listami. Rekurencyjna struktura początkowej definicji procedury `scale-list` skupia naszą uwagę na przetwarzaniu kolejnych elementów listy. Zdefiniowanie `scale-list` za pomocą `map` ukrywa przed nami takie szczegóły, a uwydatnia mnożenie przekształcające listę danych w listę wyników. Różnica między tymi dwiema definicjami nie polega na tym, że komputer wykonuje różne procesy (bo tak nie jest), ale na tym, że inaczej patrzymy na te procesy. W rezultacie `map` pomaga w ustanowieniu bariery abstrakcji oddzielającej implementację procedur przetwarzających listy od szczegółów dostępu do elementów listy i łączenia ich. Tak jak bariery pokazane na rys. 2.1, ta abstrakcja daje nam elastyczność pozwalającą na zmianę niskopoziomowych szczegółów im-

---

<sup>12</sup> Scheme standardowo udostępnia procedurę `map`, która jest bardziej ogólna niż ta opisana powyżej. Jej argumentami są procedura  $n$ -argumentowa oraz  $n$  list. `Map` stosuje tę procedurę do wszystkich pierwszych elementów list, wszystkich drugich elementów list itd., dając w wyniku listę tak uzyskanych wyników. Na przykład:

```
(map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)

(map (lambda (x y) (+ x (* 2 y)))
      (list 1 2 3)
      (list 4 5 6))
(9 12 15)
```

implementacji ciągów przy zachowaniu koncepcyjnego szkieletu operacji przekształcających ciągi. W punkcie 2.2.3 rozwijamy takie zastosowanie ciągów jako podstawy organizacji programów.

### Ćwiczenie 2.21

Argumentem procedury `square-list` jest lista liczb, a jej wynikiem jest lista kwadratów tych liczb.

```
(square-list (list 1 2 3 4))
(1 4 9 16)
```

Oto dwie różne definicje `square-list`. Uzupełnij każdą z nich, wstawiając brakujące wyrażenia:

```
(define (square-list items)
  (if (null? items)
      nil
      (cons ??? (??)))))

(define (square-list items)
  (map ??? (??)))
```

### Ćwiczenie 2.22

Ludwik Myślicielak stara się tak przepisać pierwszą procedurę `square-list` z ćwiczenia 2.21, żeby tworzyła proces iteracyjny:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

Niestety, elementy listy otrzymane w wyniku tak zdefiniowanej procedury `square-list` są ułożone w odwrotnej (niż pożądana) kolejności. Dlaczego?

Ludwik próbuje naprawić swój błąd, zamieniając miejscami argumenty `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things)))))))
  (iter items nil))
```

To też nie zadziała. Wyjaśnij, dlaczego.

### Ćwiczenie 2.23

Procedura `for-each` jest podobna do `map`. Jej argumentami są procedura i lista elementów. Jednak zamiast tworzyć listę wyników, `for-each` tylko stosuje daną procedurę kolejno, od lewej do prawej, do każdego elementu listy. Wyniki wywołań procedury nie są wcale używane — `for-each` jest stosowana do procedur wykonujących akcje, takie jak drukowanie. Na przykład:

```
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
```

Wynik wywołania `for-each` (nie pokazany powyżej) może być czymkolwiek, na przykład prawdą. Podaj implementację procedury `for-each`.

#### 2.2.2. Hierarchiczne struktury danych

Reprezentację ciągów za pomocą list można uogólnić w naturalny sposób do reprezentacji ciągów, których elementy mogą same być ciągami. Możemy na przykład traktować obiekt `((1 2) 3 4)` zbudowany przez

```
(cons (list 1 2) (list 3 4))
```

jako listę złożoną z trzech elementów, z których pierwszy sam jest listą `(1 2)`. Jest to wręcz narzucone przez postać, w jakiej interpreter wypisuje wynik. Na rysunku 2.5 jest pokazana reprezentacja tej struktury za pomocą par.

Na ciągi, których elementami są ciągi, możemy spojrzeć inaczej — jak na drzewa. Elementy ciągu są gałęziami drzewa, a elementy, które same są ciągami, są poddrzewami. Na rysunku 2.6 widać strukturę z rys. 2.5 przedstawioną jako drzewo.

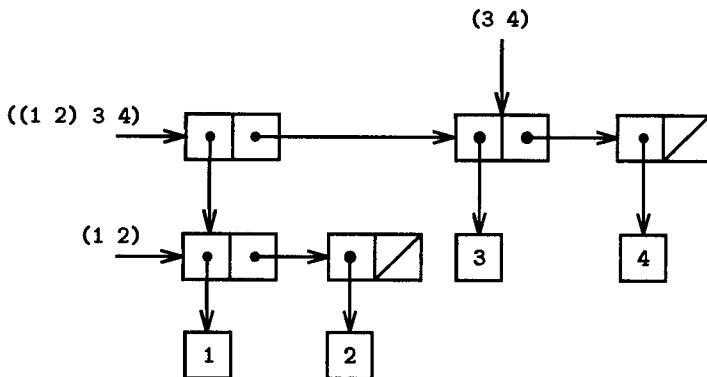
Rekursja jest naturalnym narzędziem do operowania na drzewach, gdyż często możemy sprowadzić operacje na drzewach do operacji na ich gałęziach, co z kolei sprowadza się do operacji na gałęziach gałęzi itd., aż dojdziemy do liści drzewa. Porównajmy na przykład procedurę `length` z punktu 2.2.1 z procedurą `count-leaves`, której wynikiem jest łączna liczba liści w drzewie:

```
(define x (cons (list 1 2) (list 3 4)))

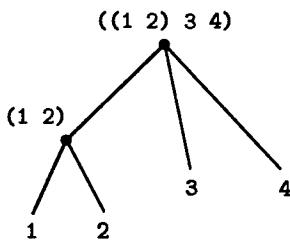
(length x)
3

(count-leaves x)
4

(list x x)
(((1 2) 3 4) ((1 2) 3 4))
```



Rys. 2.5. Struktura utworzona przez (cons (list 1 2) (list 3 4))



Rys. 2.6. Struktura listowa z rys. 2.5 przedstawiona jako drzewo

```
(length (list x x))
2

(count-leaves (list x x))
8
```

W celu zaimplementowania `count-leaves` przypomnijmy sobie rekurencyjny schemat obliczania `length`:

- Długość listy `x` jest równa 1 plus długość `cdr` listy `x`.
- Długość listy pustej wynosi 0.

Procedura `count-leaves` jest podobna. Wartość dla listy pustej jest taka sama:

- Wynikiem `count-leaves` dla listy pustej jest 0.

Jednak redukując problem, gdy odrywamy `car` listy, musimy wziąć pod uwagę, że `car` może samo być drzewem, którego liście trzeba policzyć. Tak więc odpowiedni krok redukcyjny wygląda następująco:

- Wynikiem count-leaves dla drzewa x jest wynik count-leaves dla car drzewa x plus wynik count-leaves dla cdr drzewa x.

Biorąc kolejne car, dochodzimy w końcu do liści, dla których potrzebujemy kolejnej reguły:

- Wynikiem count-leaves dla liści jest 1.

Scheme ułatwia pisanie procedur rekurencyjnych operujących na drzewach, udostępniając procedurę pierwotną pair?, która sprawdza, czy jej argument jest parą. Oto cała procedura<sup>13</sup>:

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                  (count-leaves (cdr x))))))
```

### Ćwiczenie 2.24

Założmy, że obliczamy wartość wyrażenia (list 1 (list 2 (list 3 4))). Podaj wynik, jaki wypisze interpreter, odpowiednią pudełkowo-wskaźnikową reprezentację struktury danych i jej przedstawienie w postaci drzewa (tak jak na rys. 2.6).

### Ćwiczenie 2.25

Dla każdej z podanych poniżej list podaj kombinację car i cdr, która wydobędzie z niej liczbę 7:

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7)))))))
```

### Ćwiczenie 2.26

Założmy, że zdefiniowaliśmy x i y jako takie dwie listy:

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

Co wypisze interpreter w wyniku obliczenia wartości następujących wyrażeń:

```
(append x y)
(cons x y)
(list x y)
```

<sup>13</sup> Kolejność pierwszych dwóch klauzul instrukcji cond ma znaczenie, gdyż lista pusta spełnia predykat null?, choć nie jest parą.

### Ćwiczenie 2.27

Zmień procedurę `reverse` z ćwiczenia 2.18 tak, aby powstała procedura `deep-reverse`, której argumentem jest lista, a wynikiem jest lista powstała przez jej odwrócenie i zastosowanie procedury `deep-reverse` do wszystkich jej elementów będących również listami. Na przykład:

```
(define x (list (list 1 2) (list 3 4)))

x
((1 2) (3 4))

(reverse x)
((3 4) (1 2))

(deep-reverse x)
((4 3) (2 1))
```

### Ćwiczenie 2.28

Napisz procedurę `fringe`, której argumentem jest drzewo (reprezentowane w postaci listy), a wynikiem jest lista utworzona ze wszystkich liści tego drzewa, ułożonych w kolejności od lewej do prawej. Na przykład:

```
(define x (list (list 1 2) (list 3 4)))

(fringe x)
(1 2 3 4)

(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

### Ćwiczenie 2.29

Ruchoma konstrukcja (lub krócej: konstrukcja) ma dwa ramiona: lewe i prawe. Każde z ramion to pręt określonej długości, na końcu którego jest zawieszony ciężark lub inna konstrukcja. Taką konstrukcję możemy opisać za pomocą danych złożonych (np. za pomocą list) — jej reprezentacja składa się z opisów jej dwóch ramion:

```
(define (make-mobile left right)
  (list left right))
```

Każde ramię jest reprezentowane przez swoją długość (`length`), która musi być liczbą, oraz strukturę (`structure`), która może być albo liczbą (reprezentującą wagę ciężarka), albo inną konstrukcją:

```
(define (make-branch length structure)
  (list length structure))
```

- (a) Napisz odpowiednie selektory `left-branch` i `right-branch`, których wynikami są odpowiednie ramiona konstrukcji, oraz `branch-length` i `branch-structure`, których wynikami są składowe reprezentacji ramienia.
- (b) Korzystając z tych selektorów, napisz procedurę `total-weight`, której wynikiem jest łączna waga konstrukcji.

(c) Mówimy, że konstrukcja jest *wyważona*, gdy moment siły przyłożonej do górnego lewego ramienia jest równy momentowi siły przyłożonej do górnego prawego ramienia (tzn. gdy długość lewego ramienia pomnożona przez wielkość zawieszonego na nim ciężaru jest równa odpowiedniemu iloczynowi dla prawego ramienia) oraz każda z podkonstrukcji zawieszonych na ramionach jest również wyważona. Zaprojektuj predykat sprawdzający, czy dana konstrukcja jest wyważona.

(d) Założymy, że zmienimy reprezentację konstrukcji tak, żeby konstruktory miały następującą postać:

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

Jak dużo musisz zmienić w swoich programach, aby dostosować je do nowej reprezentacji?

### Odwzorowywanie drzew

Jak procedura `map` jest potężną abstrakcją służącą do operowania na ciągach, tak procedura `map` w połączeniu z rekursją tworzą potężną abstrakcję służącą do operowania na drzewach. Na przykład procedura `scale-tree`, analogiczna do procedury `scale-list` z punktu 2.2.1, ma dwa argumenty: czynnik liczbowy i drzewo, którego liśćmi są liczby. Jej wynikiem jest drzewo, o takim samym kształcie, w którym każdą liczbę pomnożono przez podany czynnik. Rekurencyjny schemat `scale-tree` jest podobny do schematu `count-leaves`:

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                     (scale-tree (cdr tree) factor)))))

(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))
            10)
(10 (20 (30 40) 50) (60 70))
```

Inny sposób zaimplementowania `scale-tree` polega na potraktowaniu drzewa jako ciągu poddrzew i użyciu procedury `map`. Odwzorowujemy ciąg, przemnażając kolejne poddrzewa, a z wyników tworzymy listę. W najprostszym przypadku, gdy drzewo jest liściem, po prostu mnożymy je przez czynnik:

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
          (if (pair? sub-tree)
              (scale-tree sub-tree factor)
              (* sub-tree factor)))
       tree))
```

Wiele operacji na drzewach można zaimplementować za pomocą podobnej kombinacji operacji na ciągach i rekursji.

### Ćwiczenie 2.30

Zdefiniuj procedurę `square-tree` analogiczną do procedury `square-list` z ćwiczenia 2.21, ale operującą na drzewach. Oznacza to, że procedura `square-tree` powinna działać następująco:

```
(square-tree
  (list 1
    (list 2 (list 3 4) 5)
    (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Zdefiniuj `square-tree` zarówno wprost (tzn. bez korzystania z żadnych procedur wyższych rzędów), jak i za pomocą map i rekursji.

### Ćwiczenie 2.31

Dokonaj abstrakcji rozwiązania ćwiczenia 2.30 i utwórz procedurę `tree-map`. Za jej pomocą powinieneś móc zdefiniować `square-tree` następująco:

```
(define (square-tree tree) (tree-map square tree))
```

### Ćwiczenie 2.32

Zbiory możemy reprezentować jako listy różnych elementów, a zbiór wszystkich podzbiorów danego zbioru możemy reprezentować jako listę list. Jeśli na przykład nasz zbiór to `(1 2 3)`, to zbiór wszystkich jego podzbiorów jest postaci `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Uzupełnij poniższą definicję procedury tworzącej zbiór wszystkich podzbiorów danego zbioru i wyjaśnij dokładnie, jak ona działa:

```
(define (subsets s)
  (if (null? s)
    (list nil)
    (let ((rest (subsets (cdr s))))
      (append rest (map (lambda (x) (cons s x)) rest)))))
```

### 2.2.3. Ciągi jako konwencjonalne interfejsy

Omawiając pracę z danymi złożonymi, podkreśliliśmy to, jak abstrakcja danych umożliwia konstruowanie programów bez wiązania się w szczególności reprezentacją danych i jak abstrakcja daje nam elastyczność pozwalającą na eksperymentowanie z różnymi reprezentacjami. W niniejszym punkcie wprowadzimy kolejną potężną zasadę projektowania dotyczącą pracy ze strukturami danych — zastosowanie *konwencjonalnych interfejsów*.

W podrozdziale 1.3 widzieliśmy, jak za pomocą abstrakcji programów, zaimplementowanych jako procedury wyższych rzędów, można ujmować wspólne wzorce w programach operujących na danych liczbowych. Nasza zdolność

formułowania analogicznych operacji dla danych złożonych w zasadniczy sposób zależy od stylu operowania na naszych strukturach danych. Rozważmy na przykład następującą procedurę, analogiczną do procedury `count-leaves` z punktu 2.2.2, której argumentem jest drzewo i która oblicza sumę kwadratów nieparzystych liści.

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                  (sum-odd-squares (cdr tree)))))))
```

Na pierwszy rzut oka procedura ta bardzo się różni od poniższej procedury tworzącej listę złożoną ze wszystkich parzystych liczb Fibonacciego  $\text{Fib}(k)$  dla  $k$  mniejszego lub równego danej liczbie całkowitej  $n$ :

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1)))))))
  (next 0))
```

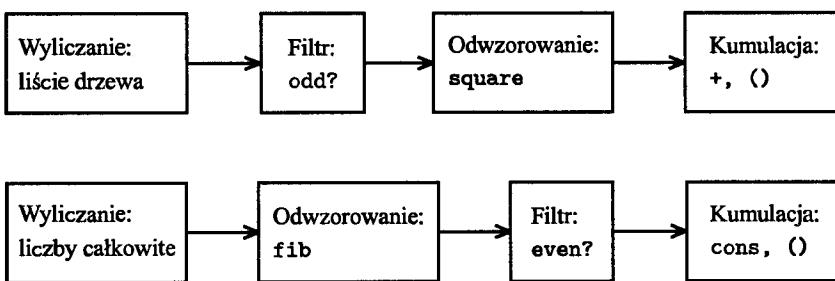
Pomimo że te dwie procedury mają zupełnie różną strukturę, bardziej abstrakcyjne opisy wykonywanych przez nie obliczeń ukazują wiele podobieństw.

### Pierwszy program

- wylicza kolejno liście drzewa;
- filtryuje je, wybierając liście nieparzyste;
- podnosi do kwadratu każdą wybraną liczbę;
- kumuluje wyniki za pomocą `+`, zaczynając od 0.

### Drugi program

- wylicza liczby całkowite od 0 do  $n$ ;
- dla każdej liczby całkowitej oblicza odpowiednią liczbę Fibonacciego;
- filtryuje uzyskane liczby, wybierając parzyste;
- kumuluje wyniki za pomocą `cons`, zaczynając od listy pustej.



Rys. 2.7. Diagramy przepływu sygnałów dla procedur `sum-odd-squares` (u góry) i `even-fibs` (na dole) ukazujące podobieństwa dwóch programów

Dla inżyniera zajmującego się przetwarzaniem sygnałów naturalne byłoby przedstawienie tych procesów jako sygnałów przepływających przez kaskadę stopni, z których każdy implementuje część programu, jak jest to pokazane na rys. 2.7. W wypadku procedury `sum-odd-squares` zaczynamy od *wyliczania* (`enumerate`), które powoduje wygenerowanie „sygnału” złożonego z liści danego drzewa. Sygnał ten przechodzi przez *filtr* (`filter`), który przepuszcza tylko elementy nieparzyste. Sygnał wynikowy przechodzi z kolei przez *odwzorowanie* (`map`), które jest „przetwornikiem” stosującym do każdego elementu procedurę `square`. Wynik odwzorowania jest następnie poddawany *kumulacji* (`accumulate`), która łączy elementy za pomocą `+`, zaczynając od wartości początkowej 0. Diagram dla procedury `even-fibs` jest analogiczny.

Niestety, podane wcześniej definicje tych dwóch procedur nie mają takiej struktury przepływu sygnałów. Jeżeli na przykład zbadamy procedurę `sum-odd-squares`, to okaże się, że wyliczanie jest zaimplementowane częściowo za pomocą predykatów `null?` i `pair?`, a częściowo przez strukturę procedury rozgałęziającej się rekurencyjnie. Podobnie, kumulacja jest realizowana częściowo przy badaniu elementów drzewa, a częściowo przy dodawaniu użytym w rekursji. Ogólnie mówiąc, w żadnej z tych procedur nie ma odrębnych części odpowiadających elementom opisu przepływu sygnałów. W naszych dwóch procedurach dekompozycja obliczeń przebiega inaczej — wyliczanie jest rozciągnięte na cały program i wymieszane z odwzorowaniem, filtrowaniem i kumulacją. Gdybyśmy mogli zorganizować nasze programy w taki sposób, żeby struktura przepływu sygnałów znalazła odbicie w pisanych przez nas procedurach, to zwiększyłoby to pojęciową przejrzystość powstającego kodu.

### Operacje na ciągach

Kluczem do organizowania programów tak, aby wyraźniej odzwierciedlały strukturę przepływu sygnałów, jest skoncentrowanie się na „sygnałach” przepływających między kolejnymi etapami procesu. Jeżeli reprezentujemy te sygnały jako listy, to możemy używać operacji na listach do zaimplementowania

ich przetwarzania na każdym z etapów. Możemy na przykład zaimplementować etapy odwzorowania z diagramów przepływu sygnałów za pomocą procedury `map` (przedstawionej w punkcie 2.2.1):

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

Przefiltrowanie ciągu w celu wybrania tylko tych elementów, które spełniają zadany predykat, można zrealizować następująco:

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence)))))
        (else (filter predicate (cdr sequence)))))
```

Na przykład:

```
(filter odd? (list 1 2 3 4 5))
(1 3 5)
```

Kumulację można zaimplementować następująco:

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(accumulate + 0 (list 1 2 3 4 5))
15

(accumulate * 1 (list 1 2 3 4 5))
120

(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)
```

Do zaimplementowania diagramów przepływu sygnałów brakuje tylko wyliczania ciągu elementów, które mają być przetwarzane. W przypadku procedury `even-fibs` potrzebujemy wygenerować ciąg liczb całkowitych z podanego zakresu, co możemy wykonać następująco:

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))
```

```
(enumerate-interval 2 7)
(2 3 4 5 6 7)
```

Do wyliczania liści drzewa możemy użyć takiej oto procedury<sup>14</sup>:

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))

(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

Teraz możemy ponownie sformułować procedury `sum-odd-squares` i `even-fibs` — zgodnie z diagramami przepływu sygnałów. W przypadku `sum-odd-squares`: wyliczamy ciąg liści drzewa; filtrujemy go, zostawiając w ciągu tylko liczby nieparzyste; każdy element podnosimy do kwadratu; sumujemy wyniki:

```
(define (sum-odd-squares tree)
  (accumulate +
              0
              (map square
                    (filter odd?
                            (enumerate-tree tree)))))
```

W przypadku `even-fibs`: wyliczamy liczby całkowite od 0 do  $n$ ; dla każdej z tych liczb obliczamy odpowiednią liczbę Fibonacciego; filtrujemy ciąg wyników, zostawiając tylko elementy parzyste; kumulujemy wyniki w postaci listy:

```
(define (even-fibs n)
  (accumulate cons
              nil
              (filter even?
                      (map fib
                            (enumerate-interval 0 n)))))
```

Znaczenie wyrażania programów jako operacji na ciągach wynika z tego, że pomaga nam w tworzeniu programów modularnych, tzn. takich programów, które powstają z połączenia względnie niezależnych fragmentów. Możemy zacheć do modularnego konstruowania programów, udostępniając biblioteki

---

<sup>14</sup> W rzeczywistości jest to dokładnie procedura `fringe` z ćwiczenia 2.28. Zmieniliśmy tutaj jej nazwę, aby podkreślić, że należy ona do rodziny ogólnych procedur operujących na ciągach.

standardowych składników wraz z konwencjonalnymi interfejsami umożliwiającymi łączenie składników na najrozmaitsze sposoby.

Projektowanie modularne to potężna strategia służąca do kontrolowania złożoności w trakcie konstruowania. Na przykład w rzeczywistych zastosowaniach przetwarzania sygnałów projektanci regularnie budują systemy, łącząc kaskadowo elementy wybrane z zestandardyzowanych rodzin filtrów i przetworników. Podobnie operacje na ciągach stanowią bibliotekę standardowych elementów programów, które możemy łączyć na najrozmaitsze sposoby. Możemy na przykład ponownie użyć fragmentów procedur `sum-odd-squares` i `even-fibs` do budowy programu tworzącego listę kwadratów pierwszych  $n + 1$  liczb Fibonacciego:

```
(define (list-fib-squares n)
  (accumulate cons
              nil
              (map square
                    (map fib
                          (enumerate-interval 0 n)))))

(list-fib-squares 10)
(0 1 1 4 9 25 64 169 441 1156 3025)
```

Możemy poprzedzać te fragmenty i użyć ich do obliczenia iloczynu kwadratów nieparzystych elementów ciągu:

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate *
              1
              (map square
                    (filter odd? sequence)))))

(product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```

Za pomocą operacji na ciągach możemy również formułować typowe programy użytkowe przetwarzające dane. Przypuśćmy, że mamy ciąg akt osobowych i chcemy wyszukać zarobki najlepiej opłacanego programisty. Założymy, że mamy dostępny selektor `salary`, którego wynikiem jest wysokość zarobków zapisana w aktach, a także predykat `programmer?` sprawdzający, czy dane akta dotyczą programisty. Możemy wówczas napisać następujący program:

```
(define (salary-of-highest-paid-programmer records)
  (accumulate max
              0
              (map salary
                    (filter programmer? records))))
```

Przykłady te ukazują tylko fragment ogromnej liczby operacji, które można wyrazić jako operacje na ciągach<sup>15</sup>.

Ciągi, zaimplementowane tutaj jako listy, służą jako konwencjonalny interfejs pozwalający na łączenie modułów przetwarzających. Dodatkowo, jeśli jednolecie stosujemy ciągi do reprezentowania struktur danych, to ograniczamy w naszych programach zależności od struktur danych do niewielkiej liczby operacji na ciągach. Zmieniając je, możemy eksperymentować z alternatywnymi reprezentacjami ciągów, nie naruszając całegoowego układu programu. Skorzystamy z tej możliwości w podrozdziale 3.5, gdzie uogólnimy paradygmat przetwarzania ciągów na ciągi nieskończone.

### Ćwiczenie 2.33

Uzupełnij następujące definicje kilku podstawowych operacji na listach, korzystających z kumulacji, wstawiając brakujące wyrażenia:

```
(define (map p sequence)
  (accumulate (lambda (x y) (??)) nil sequence))

(define (append seq1 seq2)
  (accumulate cons (??) (??)))

(define (length sequence)
  (accumulate (??) 0 sequence))
```

### Ćwiczenie 2.34

Obliczanie wartości wielomianu zmiennej  $x$  dla zadanej wartości  $x$  można sformułować za pomocą kumulacji. Wartość wielomianu

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

obliczamy, korzystając z dobrze znanego algorytmu zwanego *schematem Hornera*, który organizuje obliczenia następująco:

$$(\cdots (a_n x + a_{n-1}) x + \cdots + a_1) x + a_0$$

Innymi słowy, zaczynamy od  $a_n$ , mnożymy je przez  $x$ , dodajemy  $a_{n-1}$ , mnożymy przez  $x$  itd., aż dojdziemy do  $a_0$ <sup>16</sup>.

<sup>15</sup> Richard Waters [107] opracował program, który automatycznie analizuje tradycyjne programy w Fortranie i przedstawia je za pomocą odwzorowań, filtrów i kumulacji. Odkrył on, że aż 90% kodu Pakietu Podprogramów Naukowych Fortranu (ang. *Fortran Scientific Subroutine Package*) pasuje elegancko do tego paradygmatu. Jedną z przyczyn sukcesu Lispu jako języka programowania jest to, że listy stanowią standardowy środek wyrażania uporządkowanych kolekcji, na których można operować za pomocą operacji wyższych rzędów. Język programowania APL zawdzięcza dużo ze swojej mocy i uroku podobnemu wyborowi. W APL wszystkie dane są reprezentowane jako tablice, a dla wszystkich rodzajów operacji na tablicach jest dostępny uniwersalny i dogodny zbiór operatorów ogólnych.

<sup>16</sup> Jak podaje Knuth [59], schemat ten został sformułowany na początku XIX w. przez W. G. Hornera, jednakże był on faktycznie użyty przez Newtona już ponad sto lat wcześniej. Schemat

Uzupełnij poniższy szablon tak, aby powstała procedura obliczająca wartość wielomianu za pomocą schematu Hornera. Przyjmij, że współczynniki wielomianu są ułożone w ciągu od  $a_0$  do  $a_n$ .

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) (?))
              0
              coefficient-sequence))
```

Chcąc obliczyć na przykład  $1 + 3x + 5x^3 + x^5$  dla  $x = 2$ , powinno się wywołać

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

### Ćwiczenie 2.35

Zdefiniuj ponownie procedurę `count-leaves` z punktu 2.2.2 za pomocą kumulacji:

```
(define (count-leaves t)
  (accumulate (?) (?) (map (?) (?))))
```

### Ćwiczenie 2.36

Procedura `accumulate-n` jest podobna do `accumulate` z tym, że jej trzeci argument jest ciągiem ciągów, co do których zakładamy, że wszystkie mają tę samą liczbę elementów. Stosuje ona zadaną procedurę kumulującą do połączenia wszystkich pierwszych elementów ciągów, wszystkich drugich elementów ciągów itd., a jej wartością jest ciąg wyników. Jeśli na przykład `s` jest ciągiem zawierającym cztery ciągi:  $((1\ 2\ 3)\ (4\ 5\ 6)\ (7\ 8\ 9)\ (10\ 11\ 12))$ , to wartością procedury `(accumulate-n + 0 s)` powinien być ciąg  $(22\ 26\ 30)$ . Uzupełnij brakujące wyrażenia w następującej definicji `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init (?))
            (accumulate-n op init (?))))))
```

### Ćwiczenie 2.37

Przypuśćmy, że reprezentujemy wektory  $v = (v_i)$  jako ciągi liczb, a macierze  $m = (m_{ij})$  jako ciągi wektorów (wierszy macierzy).

Hornera wymaga mniejszej liczby dodawań i mnożeń niż prosta metoda polegająca na obliczeniu najpierw  $a_n x^n$ , następnie dodaniu do tego  $a_{n-1} x^{n-1}$  itd. W rzeczywistości można pokazać, że każdy algorytm obliczania wartości dowolnych wielomianów wymaga co najmniej tylu dodawań i mnożeń co schemat Hornera, a więc schemat Hornera jest optymalnym algorytmem obliczania wartości wielomianu. Zostało to udowodnione (dla liczby dodawań) przez A. M. Ostrowskiego w artykule z 1954 r., który w gruncie rzeczy zapoczątkował współczesne studia nad algorytmami optymalnymi. Analogiczne twierdzenie dla liczby mnożeń zostało udowodnione przez V. Y. Pana w 1966 r. W książce Borodina i Munro [9] można znaleźć przegląd tych i innych wyników dotyczących algorytmów optymalnych.

Na przykład macierz

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

jest reprezentowana jako ciąg ((1 2 3 4) (4 5 6 6) (6 7 8 9)). Przy takiej reprezentacji możemy użyć operacji na ciągach do zwięzłego wyrażenia podstawowych operacji na macierzach i wektorach. Są to następujące operacje (ich opis można znaleźć w dowolnym podręczniku algebrai macierzy):

- |                               |   |
|-------------------------------|---|
| (dot-product <i>v w</i> )     | wynikiem jest iloczyn skalarny <i>v</i> i <i>w</i> , równy $\sum_i v_i w_i$ ;                                     |
| (matrix-*-vector <i>m v</i> ) | wynikiem jest wektor <i>t</i> będący iloczynem <i>m</i> i <i>v</i> , przy czym $t_i = \sum_j m_{ij} v_j$ ;        |
| (matrix-*-matrix <i>m n</i> ) | wynikiem jest macierz <i>p</i> będąca iloczynem <i>m</i> i <i>n</i> , przy czym $p_{ij} = \sum_k m_{ik} n_{kj}$ ; |
| (transpose <i>m</i> )         | wynikiem jest macierz <i>n</i> będąca transpozycją macierzy <i>m</i> , czyli $n_{ij} = m_{ji}$ .                  |

Iloczyn skalarny możemy zdefiniować jako<sup>17</sup>

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Uzupełnij brakujące wyrażenia w poniższych procedurach obliczających pozostałe operacje na macierzach. (Procedura *accumulate-n* jest zdefiniowana w ćwiczeniu 2.36).

```
(define (matrix-*-vector m v)
  (map (??) m))

(define (transpose mat)
  (accumulate-n (??) (??) mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (??) m)))
```

### Ćwiczenie 2.38

Procedura *accumulate* jest również nazywana prawostronnym składaniem (*fold-right*), gdyż łączy pierwszy element ciągu z wynikiem złączenia wszystkich elementów na prawo od niego. Jest również lewostronne składanie (*fold-left*), podobne do *fold-right*, z tym że łączy elementy, działając w odwrotnym kierunku:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
```

---

<sup>17</sup> W definicji tej korzystamy z rozszerzonej wersji *map* opisanej w przypisie 12.

```

  result
  (iter (op result (car rest))
        (cdr rest)))
  (iter initial sequence))

```

Jakie będą wyniki wywołania procedur

```

(fold-right / 1 (list 1 2 3))
(fold-left / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left list nil (list 1 2 3))

```

Jaką własność musi mieć op, aby zagwarantować, że `fold-right` i `fold-left` będą dla wszystkich ciągów dawały takie same wyniki.

### Ćwiczenie 2.39

Uzupełnij następującą definicję procedury `reverse` (zob. ćwiczenie 2.18), korzystając z procedur `fold-right` i `fold-left` z ćwiczenia 2.38:

```

(define (reverse sequence)
  (fold-right (lambda (x y) (??)) nil sequence))

(define (reverse sequence)
  (fold-left (lambda (x y) (??)) nil
sequence))

```

### Odwzorowania zagnieżdżone

Możemy rozszerzyć paradygmat ciągów tak, aby nim objąć wiele obliczeń, które zwykle są wyrażane za pomocą pętli zagnieżdżonych<sup>18</sup>. Rozważmy taki problem: mając daną dodatnią liczbę całkowitą  $n$ , należy znaleźć wszystkie uporządkowane pary różnych dodatnich liczb całkowitych  $i$  i  $j$ , takie że  $1 \leq j < i \leq n$  oraz  $i + j$  jest liczbą pierwszą.

Jeśli na przykład  $n = 6$ , to mamy następujące pary:

$i$	2	3	4	4	5	6	6
$j$	1	2	1	3	2	1	5
$i + j$	3	5	5	7	7	7	11

Naturalny sposób zorganizowania takiego obliczenia polega na: wygenerowaniu ciągu wszystkich uporządkowanych par dodatnich liczb całkowitych nie

<sup>18</sup> Takie podejście do odwzorowań zagnieżdżonych pokazał nam David Turner, którego języki KRC i Miranda zawierają eleganckie formalizmy do operowania na takich konstrukcjach. Ćwiczenia zamieszczone w tym punkcie (zob. też ćwiczenie 2.42) pochodzą z [105]. W punkcie 3.5.3 zobaczymy, jak można uogólnić to podejście na ciągi nieskończone.

większych niż  $n$ ; przefiltrowaniu ich w celu wybrania tylko tych par, których suma jest liczbą pierwszą; utworzeniu trójki  $(i, j, i + j)$  dla każdej pary  $(i, j)$ , która przejdzie przez filtr.

Oto sposób wygenerowania ciągu par: dla każdej liczby całkowitej  $i \leq n$  wylicz liczby całkowite  $j < i$  i dla wszystkich takich  $i$  i  $j$  utwórz pary  $(i, j)$ . Używając terminologii dotyczącej operacji na ciągach, odwzorowujemy ciąg (enumerate-interval 1 n). Dla każdego  $i$  w tym ciągu odwzorowujemy ciąg (enumerate-interval 1 (- i 1)). Dla każdego  $j$  w tym drugim ciągu tworzymy parę (list i j). W ten sposób dla każdego  $i$  uzyskujemy ciąg par. Łącząc wszystkie takie ciągi dla wszystkich  $i$  (kumulując je za pomocą append), otrzymujemy wymagany ciąg par<sup>19</sup>:

```
(accumulate append
            nil
            (map (lambda (i)
                        (map (lambda (j) (list i j))
                             (enumerate-interval 1 (- i 1))))
                  (enumerate-interval 1 n)))
```

Połączenie odwzorowania i kumulacji za pomocą append pojawia się tak często w tego rodzaju programach, że wyodrębnimy je w postaci osobnej procedury:

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

Teraz filtrujemy ten ciąg, wyszukując te pary, których suma jest liczbą pierwszą. Predykat filtrujący jest wywoływany dla każdego elementu ciągu; jego argument ma postać pary i musi on wydobyć liczby całkowite tworzące parę. Tak więc predykat, który należy zastosować do każdego elementu ciągu, ma postać

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

Na koniec generujemy ciąg wyników, odwzorowując ciąg odfiltrowanych par zgodnie z następującą procedurą, która tworzy trójkę złożoną z dwóch elementów pary i ich sumy:

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

Łącząc wszystkie te kroki, otrzymujemy kompletną procedurę:

---

<sup>19</sup> Pary liczb reprezentujemy tu jako dwuelementowe listy, a nie jako pary lispowe. Tak więc „para”  $(i, j)$  jest reprezentowana jako (list i j), a nie jako (cons i j).

```
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum?
              (flatmap
               (lambda (i)
                 (map (lambda (j) (list i j))
                      (enumerate-interval 1 (- i 1))))
               (enumerate-interval 1 n)))))
```

Odwzorowania zagnieżdżone są przydatne nie tylko w przypadku ciągów przebiegających przedziały. Przypuśćmy, że chcemy wygenerować wszystkie permutacje zbioru  $S$ , tzn. wszystkie sposoby uporządkowania elementów tego zbioru. Na przykład wszystkie permutacje zbioru  $\{1, 2, 3\}$  to:  $(1, 2, 3)$ ,  $(1, 3, 2)$ ,  $(2, 1, 3)$ ,  $(2, 3, 1)$ ,  $(3, 1, 2)$  i  $(3, 2, 1)$ . Oto schemat generowania permutacji zbioru  $S$ : dla każdego elementu  $x$  należącego do  $S$  wygeneruj rekurencyjnie ciąg permutacji zbioru  $S \setminus \{x\}$ <sup>20</sup> i wstaw  $x$  na początek każdej z tych permutacji. W ten sposób dla każdego  $x$  należącego do  $S$  otrzymujemy ciąg permutacji zbioru  $S$ , zaczynających się od  $x$ . Łącząc ciągi powstałe dla wszystkich  $x$ , otrzymujemy wszystkie permutacje zbioru  $S$ <sup>21</sup>:

```
(define (permutations s)
  (if (null? s) ; zbiór pusty?
      (list nil) ; ciąg zawierający zbiór pusty
      (flatmap (lambda (x)
                  (map (lambda (p) (cons x p))
                       (permutations (remove x s))))
               s)))
```

Zwrócmy uwagę, jak taka strategia postępowania sprowadza problem wygenerowania permutacji zbioru  $S$  do problemu wygenerowania permutacji mniejszych zbiorów. Dochodzimy, w końcu, do pustej listy reprezentującej zbiór pusty. W tym przypadku generujemy `(list nil)` — ciąg zawierający tylko jeden element będący ciągiem pustym. W procedurze `permutations` korzystamy z procedury `remove`. Usuwa ona z danej listy zadawaną wartość. Można ją wyrazić w postaci prostego filtru:

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
          sequence))
```

<sup>20</sup> Zbiór  $S \setminus \{x\}$  zawiera wszystkie elementy należące do  $S$  z wyjątkiem  $x$ .

<sup>21</sup> W języku Scheme używamy średników na oznaczenie *komentarzy*. Wszystko, co napiszemy po średniku do końca wiersza, zostanie zignorowane przez interpreter. W książce tej nie używamy wielu komentarzy; staramy się, stosując opisowe nazwy, aby nasze programy same się dokumentowały.

**Ćwiczenie 2.40**

Zdefiniuj procedurę `unique-pairs`, która dla danej liczby całkowitej  $n$  generuje ciąg par  $(i, j)$ , przy czym  $1 \leq j < i \leq n$ . Użyj `unique-pairs` do uproszczenia przedstawionej powyżej definicji `prime-sum-pairs`.

**Ćwiczenie 2.41**

Napisz procedurę znajdująca wszystkie takie uporządkowane trójki różnych dodatnich liczb całkowitych  $i, j$  i  $k$ , nie większych od zadanej liczby całkowitej  $n$ , dla których suma  $i + j + k$  jest równa zadanej liczbie całkowitej  $s$ .

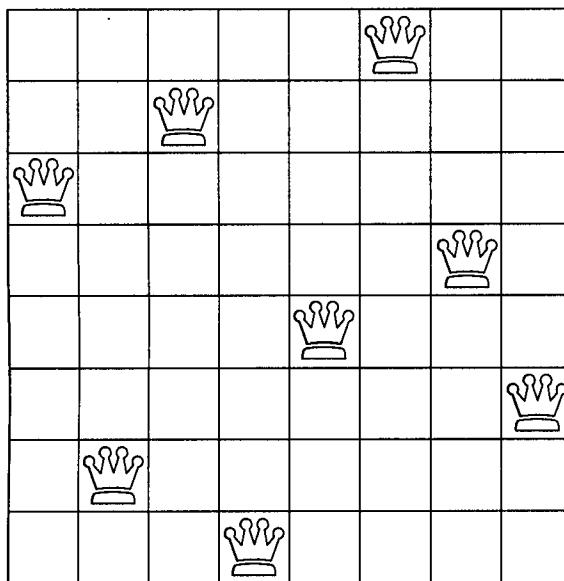
**Ćwiczenie 2.42**

Problem „ośmiu hetmanów” polega na tym, że należy rozmieścić na szachownicy osiem hetmanów tak, aby żadne dwa z nich się nie szachowały (tzn. żadne dwa z nich nie znajdowały się w tym samym wierszu, kolumnie lub przekątnej). Jedno z możliwych rozwiązań jest przedstawione na rys. 2.8. Jeden ze sposobów rozwiązywania tego problemu polega na umieszczaniu kolejnych hetmanów w kolejnych kolumnach. Gdy umieścimy już  $k - 1$  hetmanów, wówczas  $k$ -tego hetmana musimy umieścić w takim miejscu, aby nie szachował żadnego z hetmanów już ustawionych na szachownicy. Możemy takie podejście wyrazić rekurencyjnie. Założymy, że utworzyliśmy już ciąg wszystkich możliwych ustawień  $k - 1$  hetmanów w pierwszych  $k - 1$  kolumnach szachownicy. Dla każdego z tych ustawień tworzymy zbiór jego rozszerzeń polegających na dostawieniu kolejnego hetmana na każdym polu  $k$ -tej kolumny. Następnie filtrujemy je, zostawiając tylko te ustawienia, w których hetman stojący w  $k$ -tej kolumnie nie szachuje żadnego z pozostałych hetmanów. W ten sposób uzyskujemy ciąg wszystkich sposobów rozmieszczenia  $k$  hetmanów w  $k$  pierwszych kolumnach. Kontynuując tę procedurę, otrzymamy nie tylko jedno, ale wszystkie rozwiązania problemu.

Rozwiązywanie to implementujemy w postaci procedury `queens`, której wynikiem jest ciąg wszystkich rozwiązań problemu rozmieszczenia  $n$  hetmanów na szachownicy  $n \times n$ . Procedura ta zawiera procedurę wewnętrzną `queen-cols`, której wynikiem jest ciąg wszystkich sposobów rozmieszczenia  $k$  hetmanów w  $k$  pierwszych kolumnach szachownicy.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
          (lambda (positions) (safe? k positions))
          (flatmap
            (lambda (rest-of-queens)
              (map (lambda (new-row)
                      (adjoin-position new-row k rest-of-queens))
                  (enumerate-interval 1 board-size)))
            (queen-cols (- k 1))))))
    (queen-cols board-size)))
```

W procedurze tej `rest-of-queens` to sposób rozmieszczenia  $k - 1$  hetmanów w pierwszych  $k - 1$  kolumnach, a `new-row` to proponowany wiersz, w którym na-



Rys. 2.8. Rozwiązywanie problemu ośmiu hetmanów

leży ustawić hetmana stojącego w  $k$ -tej kolumnie. Uzupełnij powyższy program: zaimplementuj reprezentację zbiorów pozycji na szachownicy, wliczając w to procedurę `adjoin-position` dodającą do zbioru pozycji nową pozycję określona przez wiersz i kolumnę, a także procedurę `empty-board` reprezentującą pusty zbiór pozycji. Musisz także napisać procedurę `safe?` określającą dla danego zbioru pozycji, czy hetman stojący w  $k$ -tej kolumnie nie szachuje żadnego z pozostałych hetmanów. (Zwróć uwagę, że należy jedynie sprawdzić, czy nowy hetman nie szachuje żadnego z pozostałych — pozostałe hetmany są tak ustalone, że żadne dwa się nie szachują).

### Ćwiczenie 2.43

Ludwik Myślicielak ma duże problemy ze zrobieniem ćwiczenia 2.42. Wydaje się, że jego procedura `queens` działa, ale niezmiernie wolno. (Ludwik nigdy nie zdołał się doczekać rozwiązania dla szachownicy  $6 \times 6$ ). Gdy poprosił o pomoc Ewę Lu Ator, zwróciła mu ona uwagę, że w procedurze `flatmap` zamienił kolejność odwzorowań zagnieżdzonych:

```
(flatmap
  (lambda (new-row)
    (map (lambda (rest-of-queens)
            (adjoin-position new-row k rest-of-queens))
         (queen-cols (- k 1))))
  (enumerate-interval 1 board-size))
```

Wyjaśnij, dlaczego taka zamiana powoduje, że program działa powoli. Oszacuj, jak długo program Ludwika będzie rozwiązywał problem ośmiu hetmanów, zakładając, że program z ćwiczenia 2.42 rozwiązuje go w czasie  $T$ .

### 2.2.4. Przykład: język graficzny

W niniejszym punkcie przedstawiamy prosty język służący do rysowania (tworzenia) obrazów, ukazujący moc abstrakcji danych i własności domknięcia, a także w zasadniczy sposób korzystający z procedur wyższych rzędów. Język ten jest tak zaprojektowany, żeby można było w nim łatwo eksperymentować z takimi wzorami (jak przedstawione na rys. 2.9), które składają się z poprzesuwanych i przeskalowanych powtarzających się elementów<sup>22</sup>. W języku tym obiekty danych, które łączymy, są reprezentowane w postaci procedur, a nie struktur listowych. Tak jak `cons`, spełniający własność domknięcia, pozwala nam łatwo budować dowolnie skomplikowane struktury listowe, tak operacje tego języka, również spełniające własność domknięcia, umożliwiają nam łatwe tworzenie dowolnie skomplikowanych wzorów.

#### Język graficzny

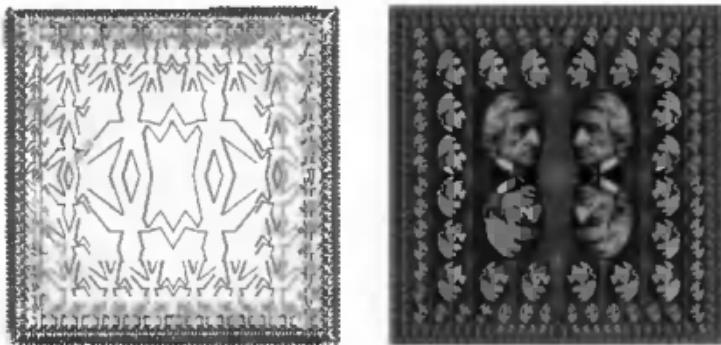
Gdy w podrozdziale 1.1 zaczynaliśmy naukę programowania, podkreślaliśmy, jak ważne jest, aby opisywać język, koncentrując się na jego elementach pierwotnych, tzn. środkach łączenia i abstrakcji. Będziemy w ten sam sposób postępowali tutaj.

Piękno języka graficznego polega częściowo na tym, że występuje w nim tylko jeden rodzaj elementów nazywanych *rysownikami* (ang. *painters*). Rysownik kreśli obraz, który jest tak przesunięty i przeskalowany, aby pasował do wyznaczonej ramki (ang. *frame*) w kształcie równoległoboku. Istnieje na przykład rysownik pierwotny, który będziemy nazywali *wave*, tworzący proste obrazy kreskowe przedstawione na rys. 2.10. Faktyczny kształt obrazu zależy od kształtu ramki — wszystkie cztery obrazy widoczne na rys. 2.10 zostały wytworzone przez ten sam rysownik *wave*, ale dla czterech różnych ramek. Rysowniki mogą być bardziej skomplikowane niż na przykład rysownik pierwotny o nazwie *rogers*, który maluje portret założyciela MIT, Williama Baronta Rogersa, jak to widać na rys. 2.11<sup>23</sup>. Cztery obrazy przedstawione na

<sup>22</sup> Język ten opiera się na języku opracowanym przez Petera Hendersona do tworzenia rysunków w stylu drzeworytu M.C. Eschera „Square Limit” (granica kwadratu) [48]. Drzeworyt ten zawiera powtarzane i przeskalowane wzory, podobne do kompozycji rysowanych przez procedurę *square-limit* przedstawioną w niniejszym punkcie.

<sup>23</sup> William Barton Rogers (1804–1882) był założycielem i pierwszym rektorem MIT. Geolog i utalentowany nauczyciel, uczył w William and Mary College oraz na University of Virginia. W 1859 r. przeniósł się do Bostonu, gdzie miał więcej czasu na badania, pracował nad planem założenia „instytutu politechnicznego” oraz zajmował stanowisko pierwszego Stanowego Inspektora Liczników Gazowych w Massachusetts.

Gdy w 1861 r. założono MIT, Rogers został wybrany na pierwszego rektora. Rogers opowiadał się za ideą „nauczania rzeczy użytecznych”, która odbiegała od edukacji uniwersyteckiej tych czasów, z jej nadmiernym naciskiem na przedmioty klasyczne, które jak pisał: „stoją na drodze ogólniejszym, wyższym i praktyczniejszym naukom i dyscyplinom przyrodniczym i społecznym”. Takie kształcenie odbiegało również od wąskiego kształcenia w szkołach handlowych. Mówiąc słowami Rogersa:



Rys. 2.9. Wzory tworzone za pomocą języka graficznego

rys. 2.11 zostały narysowane dla tych samych ramek co obrazy wytworzone przez rysownik wave na rys. 2.10.

Do łączenia obrazów używamy różnych operacji, które na podstawie danych rysowników tworzą nowe rysowniki. Na przykład operacja `beside` na podstawie dwóch rysowników tworzy nowy, złożony rysownik, który w lewej połowie ramki daje obraz tworzony przez pierwszy rysownik, a w prawej połowie ramki obraz tworzony przez drugi rysownik. Podobnie, `below` na podstawie dwóch

Narzucone na całym świecie rozróżnienie między praktykami i naukowcami jest całkowicie daremne, a całe współczesne doświadczenie wykazuje jego całkowitą bezwartościowość.

Rogers piastował stanowisko rektora MIT do 1870 r., kiedy to ustąpił z powodu złego stanu zdrowia. Drugi rektor MIT, John Runkle, w 1878 r. ustąpił pod presją kryzysu finansowego spowodowanego krachem na giełdzie w 1873 r. i ciężarem zwalczania prób przejęcia MIT przez Harvard. Rogers powrócił na stanowisko rektora do 1881 r.

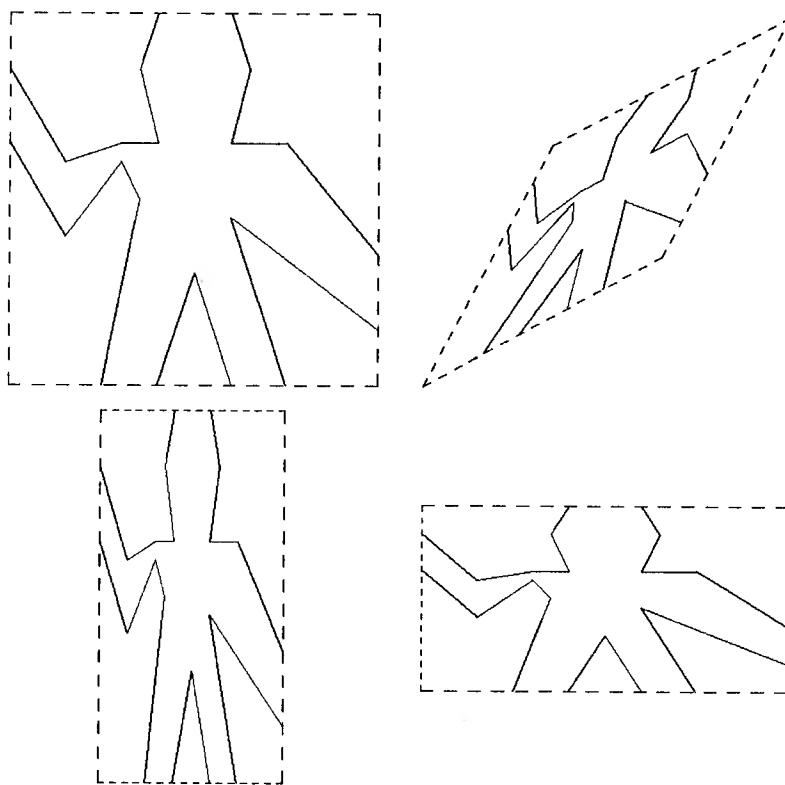
Rogers zasłabił i zmarł w trakcie przemówienia wygłoszanego na inauguracyjnych zajęciach w 1882 r. do rocznika dyplomowego. Runkle zacytował ostatnie słowa Rogersa w pamiątkowym przemówieniu wygłoszonym w tym samym roku:

„Stojąc tutaj dzisiaj i widząc, czym jest Instytut, ... przywołuję w pamięci początki nauki. Pamiętam, że sto pięćdziesiąt lat temu Stephen Hales opublikował pamphlet na temat gazu świetlnego, w którym stwierdził, iż jego badania wykazały, że 128 granów węgla bitumicznego —”

„Węgiel bitumiczny”, takie były jego ostatnie słowa na ziemi. Pochylił się do przodu, jakby coś sprawdzał w notatkach na stole przed nim, następnie powoli się wyprostował, wyrzucił ręce do góry i przeniósł się z miejsca swoich ziemskich prac i triumfów do „jutra śmierci”, gdzie tajemnice życia są rozwiązyane, a bezcierlesne duchy znajdują niekończącą się satysfakcję w kontemplowaniu nowych i ciągle niewyobrażalnych tajemnic nieskończonej przyszłości.

Mówiąc słowami Francisza A. Walkera (trzeciego rektora MIT):

Przez całe swoje życie był wierny i heroiczny, i umarł tak, jak tylko dobry rycerz mógłby tego pragnąć, w rynsztunku, na posterunku, dokładnie w trakcie spełniania publicznego obowiązku.



Rys. 2.10. Obrazy tworzone przez rysownik *wave* dla czterech różnych ramek. Ramki zaznaczone linią przerywaną nie są częścią obrazów

rysowników buduje rysownik złożony, który umieszcza obraz tworzony przez pierwszy rysownik pod obrazem tworzonym przez drugi rysownik. Niektóre operacje przekształcają jeden rysownik w celu uzyskania nowego rysownika. Na przykład operacja *flip-vert* daje w wyniku rysownik, który kreśli obraz danego rysownika odwrócony do góry nogami, a operacja *flip-horiz* tworzy rysownik, który kreśli obraz danego rysownika odwrócony z lewa na prawo.

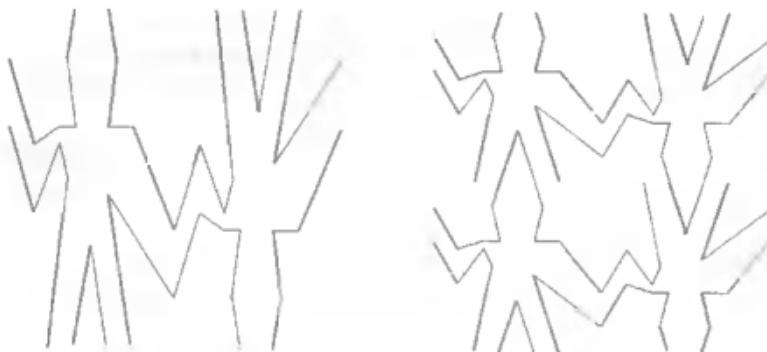
Na rysunku 2.12 widać obraz tworzony przez rysownik o nazwie *wave4* w dwóch etapach, począwszy od użycia *wave*:

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```

Montując w taki sposób obrazy złożone, korzystamy z faktu, że rysowniki mają własność domknięcia ze względu na dostępne w języku środki ich łączenia. Wynik zastosowania operacji *beside* lub *below* do dwóch rysowników jest również rysownikiem; może zatem być on użyty jako element przy budowaniu bardziej złożonych rysowników. Tak jak w przypadku tworzenia struktur

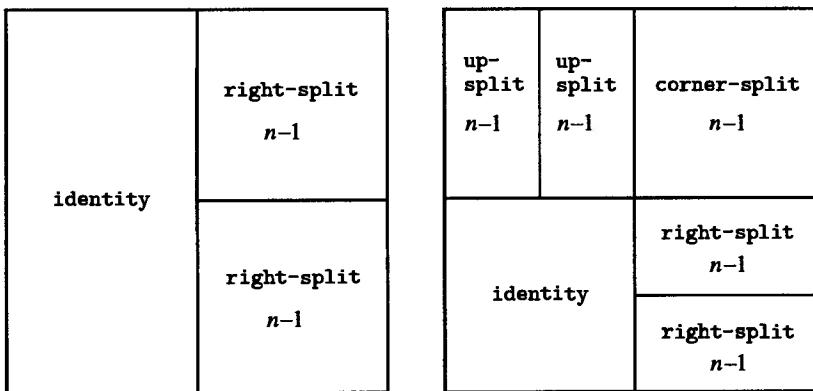


Rys. 2.11. Portrety Williama Bartona Rogersa, założyciela i pierwszego rektora MIT, namalowane dla czterech ramek, takich samych jak na rys. 2.10 (reprodukcja oryginalnego obrazu zamieszczona za zgodą MIT Museum)



```
(define wave2
  (beside wave (flip-vert wave)))
(define wave4
  (below wave2 wave2))
```

Rys. 2.12. Tworzenie złożonego obrazu, począwszy od użycia rysownika wave przedstawionego na rys. 2.10



Rys. 2.13. Wzory rekurencyjne dla operacji right-split i corner-split

listowych za pomocą `cons`, własność domknięcia naszych danych ze względu na dostępne środki ich łączenia rozstrzyga o możliwości tworzenia struktur złożonych za pomocą jedynie kilku operacji.

Skoro potrafimy łączyć rysowniki, chcielibyśmy móc tworzyć abstrakcje typowych wzorów połączonych rysowników. Zaimplementujemy operacje na rysownikach jako procedury języka Scheme. Oznacza to, że nie potrzebujemy w języku graficznym żadnego specjalnego mechanizmu abstrakcji — ponieważ środki łączenia mają postać zwykłych procedur języka Scheme, więc automatycznie możemy z operacjami na rysownikach robić wszystko to, co można robić z procedurami. Możemy na przykład utworzyć abstrakcję wzoru użytego w `wave4`:

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

i zdefiniować `wave4` jako szczególny przypadek tego wzoru:

```
(define wave4 (flipped-pairs wave))
```

Możemy również definiować operacje rekurencyjne. Oto taka operacja, dzieląca rysowniki i rozgałęzająca się po prawej stronie, jak to jest pokazane na rys. 2.13 i 2.14:

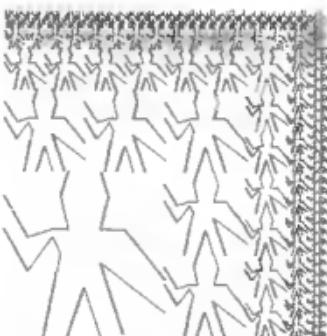
```
(define (right-split painter n)
  (if (= n 0)
    painter
    (let ((smaller (right-split painter (- n 1)))))
      (beside painter (below smaller smaller)))))
```



(right-split wave 4)



(right-split rogers 4)



(corner-split wave 4)



(corner-split rogers 4)

Rys. 2.14. Operacje rekurencyjne `right-split` i `corner-split` zastosowane do rysowników `wave` i `rogers`. Łącząc cztery obrazy tworzone przez `corner-split`, otrzymujemy symetryczne wzory, takie jak widać na rys. 2.9

Mozemy też otrzymywać przyjemne dla oka wzory, tworząc rozgałęzienia zarówno do góry, jak i w prawo (zob. ćwiczenie 2.44 oraz rys. 2.13 i 2.14):

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right corner)))))))
```

Łącząc odpowiednio cztery egzemplarze `corner-split`, otrzymujemy operację (wzór) o nazwie `square-limit`, której zastosowanie do `wave` i `rogers` jest pokazane na rys. 2.9:

```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

### Ćwiczenie 2.44

Zdefiniuj procedurę `up-split` używaną w `corner-split`. Jest ona podobna do `right-split`, z tym że `below` i `beside` są zamienione rolami.

### Operacje wyższych rzędów

Oprócz tworzenia abstrakcji wzorów połączonych rysowników możemy pracować na wyższym poziomie, tworząc abstrakcje wzorów połączonych operacji na rysownikach. Oznacza to, że możemy traktować operacje na rysownikach jak elementy, na których operujemy, i możemy określać środki łączenia takich elementów — procedury, których argumentami są operacje na rysownikach, a wynikami są nowe operacje na rysownikach.

Na przykład zarówno `flipped-pairs`, jak i `square-limit` układają kwadratowy wzór złożony z czterech egzemplarzy obrazu rysownika. Różni je przy tym jedynie sposób kompozycji obrazów. Abstrakcję takiego wzoru łączenia rysowników możemy zapisać w postaci następującej procedury, która ma cztery argumenty będące jednoargumentowymi operacjami na rysownikach i której wynikiem jest operacja przekształcająca dany rysownik za pomocą tych czterech argumentów i układająca z wyników kwadrat. `Tl`, `tr`, `bl` i `br` to przekształcenia, które należy zastosować, odpowiednio, do górnego lewego, górnego prawego, dolnego lewego i dolnego prawego egzemplarza.

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

Wówczas można zdefiniować `flipped-pairs` przy użyciu `square-of-four` w taki oto sposób<sup>24</sup>:

---

<sup>24</sup> Definicję tę moglibyśmy zapisać równoważnie jako

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                    identity flip-vert)))
    (combine4 painter)))
```

a `square-limit` można zapisać jako<sup>25</sup>

```
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-horiz identity
                                    rotate180 flip-vert)))
    (combine4 (corner-split painter n))))
```

### Ćwiczenie 2.45

`Right-split` i `up-split` można wyrazić jako szczególne przypadki ogólnej operacji dzielącej. Napisz taką procedurę `split`, dla której definicje

```
(define right-split (split beside below))
(define up-split (split below beside))
```

określają procedury `right-split` i `up-split` o takim samym działaniu jak te już zdefiniowane.

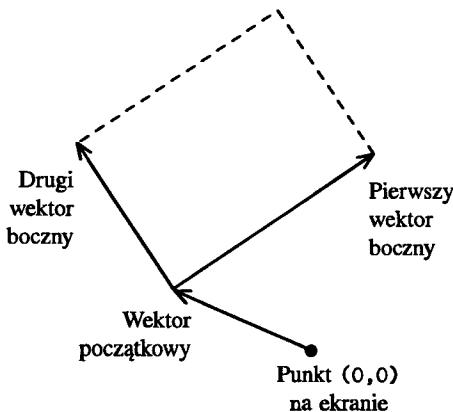
### Ramki

Zanim będziemy mogli pokazać, jak można zaimplementować rysowniki i środki ich łączenia, musimy najpierw przyjrzeć się ramkom. Ramkę można opisać za pomocą trzech wektorów — wektora początkowego (ang. *origin vector*) i dwóch wektorów bocznych (ang. *edge vectors*). Wektor początkowy określa przesunięcie „początkowego” wierzchołka ramki względem pewnego przyjętego początku układu współrzędnych na płaszczyźnie, a wektory boczne określają położenie pozostałych wierzchołków ramki względem jej początkowego wierzchołka. Jeśli wektory boczne są prostopadłe, to ramka będzie miała kształt prostokąta. W przeciwnym razie ramka będzie równoległobokiem.

Na rysunku 2.15 widać ramkę i związane z nią wektory. Zgodnie z zasadą abstrakcji danych nie musimy na razie nic zakładać o reprezentacji ramek oprócz tego, że dostępne są konstruktor `make-frame`, który z trzech wektorów tworzy opisywaną przez nie ramkę, oraz trzy odpowiednie selektory: `origin-frame`, którego wynikiem jest wektor początkowy, oraz `edge1-frame` i `edge2-frame`, których wynikami są wektory boczne (zob. ćwiczenie 2.47).

W odniesieniu do obrazów będziemy używać współrzędnych leżących w obrębie kwadratu jednostkowego ( $0 \leq x, y \leq 1$ ). Z każdą ramką wiążemy *odwzo-*

<sup>25</sup> `Rotate180` obraca rysownik o 180 stopni (zob. ćwiczenie 2.50). Zamiast `rotate180` moglibyśmy napisać (`compose flip-vert flip-horiz`), używając procedury `compose` z ćwiczenia 1.42.



Rys. 2.15. Ramkę opisują trzy wektory — wektor początkowy i dwa wektory boczne

rowanie współrzędnych ramki, które będziemy wykorzystywać do przesuwania i skalowania obrazów, tak aby pasowały do ramki. Odwzorowanie takie przekształca kwadrat jednostkowy na ramkę, przyporządkowując wektorowi  $\mathbf{v} = (x, y)$  sumę wektorów:

$$\text{początek(ramka)} + x \cdot \text{bok}_1(\text{ramka}) + y \cdot \text{bok}_2(\text{ramka})$$

Na przykład wektor  $(0, 0)$  jest odwzorowywany na wierzchołek początkowy,  $(1, 1)$  na wierzchołek leżący po przekątnej względem wierzchołka początkowego, a  $(0.5, 0.5)$  na środek ramki. Odwzorowanie współrzędnych ramki możemy zapisać w postaci następującej procedury<sup>26</sup>:

```
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
      (origin-frame frame)
      (add-vect (scale-vect (xcor-vect v)
                            (edge1-frame frame))
                (scale-vect (ycor-vect v)
                            (edge2-frame frame))))))
```

Zwróćmy uwagę, że wynikiem zastosowania `frame-coord-map` do ramki jest procedura, która przekształca wektory na wektory. Jeżeli wektor będący argumentem leży wewnątrz kwadratu jednostkowego, to wektor wynikowy będzie leżał wewnątrz ramki. Na przykład wywołanie

<sup>26</sup> Procedura `frame-coord-map` korzysta z operacji na wektorach opisanych w ćwiczeniu 2.46. Zakładamy, że operacje te zaimplementowano na podstawie pewnej reprezentacji wektorów. Ze względu na zasadę abstrakcji danych, dopóki operacje na wektorach działają poprawnie, dopóty nie ma znaczenia, jaka jest ta reprezentacja.

```
((frame-coord-map a-frame) (make-vect 0 0))
```

daje taki sam wynik co wywołanie

```
(origin-frame a-frame)
```

### Ćwiczenie 2.46

Dwuwymiarowy wektor  $\mathbf{v}$  zaczepiony w początku układu współrzędnych można reprezentować jako parę złożoną ze współrzędnych  $x$  i  $y$  jego końca. Zaimplementuj abstrakcję danych dla wektorów, definiując konstruktor `make-vect` i odpowiadające mu selektory — `xcor-vect` (podający współrzędną  $x$ ) i `ycor-vect` (podający współrzędną  $y$ ). Na podstawie swoich selektorów i konstruktorów zaimplementuj procedury `add-vect`, `sub-vect` i `scale-vect` wykonujące, odpowiednio, operacje dodawania wektorów, odejmowania wektorów i mnożenia wektora przez skalar:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

$$(x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2)$$

$$s \cdot (x, y) = (sx, sy)$$

### Ćwiczenie 2.47

Możliwe są dwa konstruktory ramek:

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
```

```
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

Dla obydwu konstruktorów napisz odpowiednie selektory, tworząc implementację ramek.

### Rysowniki

Rysownik jest reprezentowany przez procedurę, która dla danej ramki tworzy określony obraz, przesunięty i przeskalowany tak, aby pasował do ramki. Oznacza to, że jeśli  $p$  jest rysownikiem, a  $f$  jest ramką, to wywołując  $p$  z argumentem  $f$ , tworzymy obraz rysownika  $p$  w ramce  $f$ .

Szczegóły implementacji rysowników pierwotnych zależą od konkretnej charakterystyki systemu graficznego i od rodzaju tworzonego obrazu. Założymy na przykład, że jest dostępna procedura `draw-line` rysująca na ekranie odcinek łączący dwa określone punkty. Możemy wówczas w następujący sposób tworzyć rysowniki kreślące obrazy kreskowe, takie jak rysownik `wave` przedstawiony na rys. 2.10, na podstawie listy odcinków<sup>27</sup>:

<sup>27</sup> Procedura `segments->painter` korzysta z reprezentacji odcinków opisanej w ćwiczeniu 2.48. Korzysta także z procedury `for-each` opisanej w ćwiczeniu 2.23.

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        (draw-line
          ((frame-coord-map frame) (start-segment segment))
          ((frame-coord-map frame) (end-segment segment))))
      segment-list)))
```

Końce odcinków są określone za pomocą współrzędnych odnoszących się do kwadratu jednostkowego. Dla każdego odcinka z listy rysownik przekształca końce odcinka zgodnie z odwzorowaniem współrzędnych ramki i kreśli odcinek łączący otrzymane punkty.

Reprezentacja rysowników w postaci procedur tworzy w języku graficznym bardzo mocną barierę abstrakcji. Możemy tworzyć i mieszać ze sobą wszystkie rodzaje rysowników pierwotnych, opartych na najrozmaitszych możliwościach graficznych. Szczegóły ich implementacji nie mają znaczenia. Rysownikiem może być dowolna procedura, pod warunkiem że jej argumentem jest ramka i że rysuje ona coś dopasowanego do tej ramki<sup>28</sup>.

### Ćwiczenie 2.48

Odcinek skierowany na płaszczyźnie może być reprezentowany jako para wektorów — wektor prowadzący z początku układu współrzędnych do początku odcinka i wektor prowadzący z początku układu współrzędnych do końca odcinka. Wykorzystaj swoją reprezentację wektorów z ćwiczenia 2.46 do zdefiniowania reprezentacji odcinków z konstruktorem `make-segment` oraz selektorami `start-segment` i `end-segment`.

### Ćwiczenie 2.49

Użyj procedury `segments->painter` do zdefiniowania następujących rysowników pierwotnych:

- (a) rysownika kreślącego kontury zadanej ramki;
- (b) rysownika kreślącego „X” przez połączenie wierzchołków ramki leżących po przekątnej;
- (c) rysownika kreślącego równoległobok powstały przez połączenie środków boków ramki;
- (d) rysownika `wave`.

<sup>28</sup> Na przykład rysownik `rogers` przedstawiony na rys. 2.11 utworzono na podstawie czarno-białego obrazu ze stopniami szarości. Dla każdego punktu wewnętrz ramki rysownik `rogers` określa punkt na obrazie, który jest odwzorowywany na niego przez odwzorowanie współrzędnych ramki, i nadaje mu odpowiedni stopień jasności. Dopuszczając różne rodzaje rysowników, czerpiemy korzyści z zasady abstrakcji danych, opisanej w punkcie 2.1.3, gdzie przekonywaliśmy, że reprezentacja liczb wymiernych mogłaby być dowolna, byleby spełniała odpowiednie warunki. Wykorzystujemy tutaj fakt, że rysownik może być zaimplementowany w dowolny sposób, o ile tylko rysuje coś w wyznaczonej ramce. W punkcie 2.1.3 pokazaliśmy, że pary mogą być również reprezentowane jako procedury. Rysowniki są drugim przykładem proceduralnej reprezentacji danych.

## Przekształcanie i łączenie rysowników

Operacja na rysownikach (taka jak `flip-vert` czy `beside`) tworzy rysownik, który wywołuje oryginalne rysowniki dla ramek otrzymanych na podstawie ramki będącej jego argumentem. Tak więc na przykład `flip-vert` nie musi wiedzieć, jak działa jakiś rysownik, aby obrócić go do góry nogami — wystarczy, że wie, jak obrócić ramkę; obrócony rysownik zaś używa oryginalnego rysownika, ale dla obróconej ramki.

Operacje na rysownikach opierają się na procedurze `transform-painter`, której argumentami są rysownik i informacja, jak należy przekształcić ramkę, a wynikiem jest nowy rysownik. Przekształcony rysownik, gdy wywołamy go dla danej ramki, przekształca ją i wywołuje oryginalny rysownik dla przekształconej ramki. Argumentami `transform-painter` są punkty (reprezentowane przez wektory) określające wierzchołki nowej ramki: gdy odwzorujemy je na daną ramkę, wtedy pierwszy punkt określa wierzchołek początkowy nowej ramki, a pozostałe dwa określają końce jej wektorów bocznych. Tak więc argumenty leżące w kwadracie jednostkowym określają ramkę mieszczącą się w oryginalnej ramce.

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
          (make-frame new-origin
                      (sub-vect (m corner1) new-origin)
                      (sub-vect (m corner2) new-origin)))))))
```

Oto jak możemy odwracać obrazy rysowników w pionie:

```
(define (flip-vert painter)
  (transform-painter painter
    (make-vect 0.0 1.0) ; nowy origin
    (make-vect 1.0 1.0) ; nowy koniec edge1
    (make-vect 0.0 0.0))) ; nowy koniec edge2
```

Korzystając z `transform-painter`, możemy z łatwością definiować nowe przekształcenia. Możemy na przykład zdefiniować rysownik, który zmniejsza swój obraz do rozmiarów górnej prawej ćwiartki zadanej ramki:

```
(define (shrink-to-upper-right painter)
  (transform-painter painter
    (make-vect 0.5 0.5)
    (make-vect 1.0 0.5)
    (make-vect 0.5 1.0)))
```

lub przekształcenie powodujące obrót obrazu o 90 stopni w kierunku przeciwnym do ruchu wskazówek zegara<sup>29</sup>:

```
(define (rotate90 painter)
  (transform-painter painter
    (make-vect 1.0 0.0)
    (make-vect 1.0 1.0)
    (make-vect 0.0 0.0)))
```

czy też przekształcenie powodujące ściśnięcie obrazu w kierunku środka ramki<sup>30</sup>:

```
(define (squash-inwards painter)
  (transform-painter painter
    (make-vect 0.0 0.0)
    (make-vect 0.65 0.35)
    (make-vect 0.35 0.65)))
```

Przekształcenia ramek są również kluczem do definiowania sposobów łączenia dwóch lub więcej rysowników. Procedura `beside` na przykład ma dwa argumenty będące rysownikami, przekształca je tak, aby rysowały odpowiednio w lewej i prawej połowie danej ramki, i tworzy nowy złożony rysownik. Gdy złożony rysownik otrzyma ramkę, wywołuje pierwszy rysownik przekształcony tak, aby rysował w lewej połówce ramki, a następnie wywołuje drugi rysownik przekształcony tak, aby rysował w prawej połówce ramki:

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
            (transform-painter painter1
              (make-vect 0.0 0.0)
              split-point
              (make-vect 0.0 1.0)))
          (paint-right
            (transform-painter painter2
              split-point
              (make-vect 1.0 0.0)
              (make-vect 0.5 1.0))))
      (lambda (frame)
        (paint-left frame)
        (paint-right frame)))))
```

<sup>29</sup> `Rotate90` dokonuje czystego obrotu tylko dla kwadratowej ramki, gdyż rozciąga i ścięśnia obraz tak, aby pasował do obróconej ramki.

<sup>30</sup> Obrazy w kształcie rombów na rys. 2.10 i 2.11 powstały przez zastosowanie `squash-inwards` do `wave` i `rogers`.

Zwróćmy uwagę, jak abstrakcja danych rysowników, a w szczególności reprezentacja rysowników w postaci procedur, sprawia, że `beside` jest łatwe do zaimplementowania. Procedura `beside` nie musi nic wiedzieć o szczegółach składowych rysowników oprócz tego, że każdy rysownik będzie tworzył jakiś obraz w wyznaczonej ramce.

### Ćwiczenie 2.50

Zdefiniuj przekształcenie `flip-horiz` powodujące odwrócenie rysowników w poziomie oraz przekształcenia powodujące obrót rysowników o 180 i 270 stopni w kierunku przeciwnym do ruchu wskazówek zegara.

### Ćwiczenie 2.51

Zdefiniuj dla rysowników operację `below`. Operacja ta ma dwa argumenty będące rysownikami. Jej wynikiem jest rysownik, który dla zadanej ramki używa pierwszego z danych rysowników w dolnej połowie ramki, a drugiego w górnej połowie ramki. Zdefiniuj `below` na dwa sposoby — najpierw pisząc procedurę analogiczną do procedury `beside` podanej powyżej, a następnie korzystając z `beside` i odpowiednich obrotów (z ćwiczenia 2.50).

## Poziomy języka przy konstruowaniu elastycznych programów

Język graficzny korzysta z niektórych zasadniczych koncepcji, jakie wprawdziliśmy odnośnie do abstrakcji procedur i danych. Podstawowe abstrakcje danych — rysowniki — są zaimplementowane za pomocą reprezentacji proceduralnej, która pozwala językowi na traktowanie różnych podstawowych możliwości graficznych w jednolity sposób. Metody łączenia mają własność domknięcia, co umożliwia łatwe budowanie konstrukcji złożonych. Wreszcie, wszystkie narzędzia do tworzenia abstrakcji procedur są dostępne do tworzenia abstrakcji środków łączenia rysowników.

Ujrzeliśmy również przelotnie inną zasadniczą koncepcję dotyczącą języków i konstruowania programów. Jest to podejście polegające na *projektowaniu wielopoziomowym* (ang. *stratified design*), w którym system złożony powinien składać się z szeregu poziomów opisanych za pomocą szeregu języków. Każdy poziom jest tworzony przez łączenie elementów uznawanych za pierwotne na tym poziomie, a elementy utworzone na danym poziomie są używane jako pierwotne na kolejnym poziomie. Język używany na każdym poziomie projektu wielopoziomowego zawiera pojęcia pierwotne, środki łączenia i środki abstrakcji właściwe dla jego poziomu szczegółowości.

Projektowanie wielopoziomowe przeniknęło inżynierię systemów złożonych. W inżynierii systemów komputerowych na przykład rezistory i tranzystory łączy się (i opisuje, używając języka układów analogowych) w takie elementy, jak bramki AND i OR, które stanowią pierwotne elementy języka projekto-

wania układów cyfrowych<sup>31</sup>. Te elementy łączy się, budując procesory, magistrale i systemy pamięci, które z kolei łączy się w komputery, używając języków odpowiednich dla architektury komputera. Komputery łączy się w systemy rozproszone, używając języków odpowiednich do opisu połączeń sieciowych itd.

Maleńkim przykładem wielopoziomowości może być nasz język graficzny korzystający z elementów pierwotnych (rysowników pierwotnych), które utworzono przy użyciu języka operującego punktami i odcinkami, opisującego listy odcinków dla `segments->painter` lub szczegóły cieniowania dla takich rysowników jak `rogers`. Nasz opis języka graficznego dotyczył głównie łączenia tych elementów pierwotnych za pomocą geometrycznych łączników, takich jak `beside` czy `below`. Pracowaliśmy również na wyższym poziomie, rozpatrując `beside` i `below` jako elementy pierwotne, którymi możemy manipulować w języku udostępniającym takie operacje jak `square-of-four`, wyrażające typowe wzory łączenia geometrycznych łączników.

Projektowanie wielopoziomowe pomaga w tworzeniu *elastycznych* programów, tzn. takich, dla których niewielkie zmiany w specyfikacji wymagają zwykle odpowiednio małych zmian w programie. Założymy na przykład, że chcemy zmienić obraz oparty na `wave`, pokazany na rys. 2.9. Moglibyśmy pracować na poziomie najniższym, zmieniając szczegółowy wygląd elementu `wave`; moglibyśmy pracować na poziomie średnim, zmieniając sposób, w jaki `corner-split` powiela `wave`; moglibyśmy też pracować na poziomie najwyższym, zmieniając sposób, w jaki `square-limit` aranżuje cztery narożniki. Ogólnie mówiąc, każdy poziom projektu wielopoziomowego dostarcza innego słownictwa do wyrażania cech systemu i innych możliwości do ich zmiany.

### Ćwiczenie 2.52

Wprowadź zmiany do „granicy kwadratu” opartej na `wave` i pokazanej na rys. 2.9 na każdym z opisanych powyżej poziomów. W szczególności:

- dodaj kilka odcinków do rysownika pierwotnego `wave` z ćwiczenia 2.49 (np. dodaj uśmiech);
- zmień wzór tworzony przez `corner-split` (tak, aby np. występowało w nim po jednym egzemplarzu obrazów `up-split` i `right-split` zamiast dwóch);
- zmień wersję `square-limit` korzystającą ze `square-of-four` tak, aby układała ćwiartki obrazu w inny sposób (np. możesz sprawić, aby wielki pan `Rogers` wyglądał na zewnątrz z każdego rogu kwadratu).

## 2.3. Dane symboliczne

Wszystkie złożone obiekty danych, z których dotychczas korzystaliśmy, były ostatecznie zbudowane z liczb. W niniejszym podrozdziale rozszerzymy zdol-

<sup>31</sup> Język taki jest opisany w punkcie 3.3.4.

ności reprezentacyjne naszego języka, wprowadzając możliwość pracowania na dowolnych symbolach jako danych.

### 2.3.1. Cytowanie

Jeśli możemy tworzyć dane złożone, używając symboli, to możemy mieć takie listy, jak

```
(a b c d)
(23 45 17)
((Norah 12) (Molly 9) (Anna 7) (Lauren 6) (Charlotte 4))
```

Listy zawierające symbole mogą wyglądać całkiem jak wyrażenia naszego języka:

```
(* (+ 23 45) (+ x 9))
(define (fact n) (if (= n 1) 1 (* n (fact (- n 1)))))
```

Aby móc operować na symbolach, potrzebujemy w naszym języku nowego elementu: możliwości *cytowania* obiektów danych. Przypuśćmy, że chcemy utworzyć listę (a b). Nie możemy tego zrobić, pisząc (list a b), gdyż takie wyrażenie tworzy listę złożoną z *wartości* a i b, a nie samych symboli. Ta kwestia jest dobrze znana w kontekście języków naturalnych, gdzie słowa i zdania mogą być rozpatrywane albo jako jednostki semantyczne, albo jako ciągi znaków (jednostki syntaktyczne). W językach naturalnych powszechnym zwyczajem jest używanie cudzysłowów na oznaczenie słów lub zdań, które należy traktować dosłownie jako ciągi znaków. Na przykład pierwszą literą słowa „John” jest oczywiście „J”. Jeśli poprosimy kogoś „powiedz głośno swoje imię”, to spodziewamy się usłyszeć imię tej osoby. Jeśli jednak poprosimy „powiedz głośno »swoje imię«”, to spodziewamy się usłyszeć słowa „swoje imię”. Zwróćmy uwagę, że musimy zagnieździć cudzysłowy, aby opisać, co ktoś inny może powiedzieć<sup>32</sup>.

W ten sam sposób możemy zaznaczać listy i symbole, które trzeba traktować jako obiekty danych, a nie wyrażenia, których wartość należy obliczyć. Jed-

<sup>32</sup> Możliwość cytowania niszczy możliwości wnioskowania o języku w prostych kategoriach, gdyż znosi zasadę mówiącą, że jedne elementy możemy zastępować równymi im innymi elementami. Na przykład trzy jest równe jeden plus dwa, ale słowo „trzy” nie jest określeniem „jeden plus dwa”. Cytowanie jest potężną techniką, gdyż otwiera możliwości budowania wyrażeń operujących na innych wyrażeniach (jak zobaczymy w rozdziale 4, gdy będziemy pisać interpreter). Dopuszczenie, żeby jedne stwierdzenia w języku dotyczyły innych stwierdzeń w tym samym języku, sprawia, że bardzo trudno utrzymać spójną zasadę mówiącą, co to znaczy, że „elementy możemy zastępować równymi im elementami”. Jeśli na przykład wiemy, że gwiazda wieczorna i gwiazda poranna to to samo, to ze stwierdzenia „gwiazda wieczorna to Wenus” możemy wywnioskować, że „gwiazda poranna to Wenus”. Jednakże z tego, że „John wie, że gwiazda wieczorna to Wenus” nie możemy wywnioskować, że „John wie, że gwiazda poranna to Wenus”.

nakże stosowana przez nas forma cytowania różni się od przyjętej w językach naturalnych tym, że umieszczamy znak cudzysłownu (tradycyjnie znak apostrofu ') tylko przed cytowanym obiektem. To nam wystarczy, gdyż w składni języka Scheme znaki odstępu i nawiasy ograniczają obiekty. Tak więc apostrof oznacza, że cytujemy obiekt, który po nim występuje<sup>33</sup>.

Możemy teraz rozróżnić symbole i ich wartości:

```
(define a 1)
(define b 2)
(list a b)
(1 2)

(list 'a 'b)
(a b)

(list 'a b)
(a 2)
```

Cytowanie pozwala nam również wprowadzać obiekty złożone, stosując tradycyjną formę zapisu list<sup>34</sup>:

```
(car '(a b c))
a

(cdr '(a b c))
(b c)
```

Dzięki temu możemy uzyskać listę pustą, obliczając wartość '() i obywając się przy tym bez zmiennej nil.

Dodatkową operacją używaną w odniesieniu do symboli jest predykat eq?, który określa, czy dwa dane symbole są takie same<sup>35</sup>. Korzystając z eq?,

<sup>33</sup> Apostrof różni się od znaku cudzysłownu, którym otaczaliśmy wypisywane napisy. Podczas gdy apostrofu można używać do oznaczania list lub symboli, znaków cudzysłownu możemy używać tylko do oznaczania napisów. W niniejszej książce używamy napisów wyłącznie jako elementów wypisywanych.

<sup>34</sup> Ścisłe rzeczą biorąc, takie zastosowanie apostrofu łamie ogólną zasadę, że wszystkie obiekty złożone powinny w naszym języku być ujęte w nawiasy i wyglądać jak listy. Możemy zachować spójność, wprowadzając formę specjalną quote, służącą do tego samego celu co apostrof. Tak więc zamiast 'a pisalibyśmy (quote a), a zamiast '(a b c) pisalibyśmy (quote (a b c)). Dokładnie tak działa interpreter. Apostrof jest tylko jednoznakowym skrótem oznaczającym objęcie przez quote następującego po nim pełnego wyrażenia i utworzenie (quote <wyrażenie>). Jest to ważne, gdyż zachowuje zasadę, że każde wyrażenie widziane przez interpreter może być traktowane jako obiekt danych. Moglibyśmy na przykład zbudować wyrażenie (car '(a b c)), które jest tym samym co (car (quote (a b c))), obliczając wartość (list 'car (list 'quote '(a b c))).

<sup>35</sup> Możemy przyjąć, że dwa symbole są „takie same”, jeśli składają się z takich samych znaków występujących w tej samej kolejności. Taka definicja nie obejmuje bardzo istotnej kwestii, do

możemy zaimplementować użyteczną procedurę nazywaną `memq`. Ma ona dwa argumenty: symbol i listę. Jeśli symbol nie występuje na liście (tzn. nie jest `eq?` żadnemu elementowi tej listy), to wynikiem procedury `memq` jest fałsz. W przeciwnym razie jej wynikiem jest podlista danej listy, zaczynająca się od miejsca pierwszego wystąpienia symbolu:

```
(define (memq item x)
  (cond ((null? x) false)
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

Na przykład wartością

```
(memq 'apple '(pear banana prune))
```

jest fałsz, podczas gdy wartością

```
(memq 'apple '(x (apple sauce) y apple pear))
```

jest `(apple pear)`.

### Ćwiczenie 2.53

Co wypisze interpreter w wyniku obliczenia wartości każdego z poniższych wyrażeń?

```
(list 'a 'b 'c)
(list (list 'george))
(cdr '((x1 x2) (y1 y2)))
(cadr '((x1 x2) (y1 y2)))
(pair? (car '(a short list)))
(memq 'red '((red shoes) (blue socks)))
(memq 'red '(red shoes blue socks))
```

### Ćwiczenie 2.54

Mówimy, że dwie listy są równe (`equal?`), jeśli zawierają takie same elementy ułożone w tej samej kolejności. Na przykład

```
(equal? '(this is a list) '(this is a list))
```

jest prawdziwe, ale

```
(equal? '(this is a list) '(this (is a) list))
```

---

rozoważenia której nie jesteśmy jeszcze gotowi — co to znaczy w języku programowania, że coś jest „takie samo”. Wróćmy do tego w rozdziale 3 (punkt 3.1.3).

jest fałszywe. Ścisłej, możemy zdefiniować `equal?` rekurencyjnie, korzystając z podstawowej równości symboli `eq?`; powiemy wtedy, że `a` i `b` są równe (`equal?`), jeśli obydwa są symbolami i spełniają `eq?` lub obydwa są listami i (`car a`) jest równe (`car b`), a (`cdr a`) jest równe (`cdr b`). Opierając się na tym pomyśle, zaimplementuj predykat `equal?` w postaci procedury<sup>36</sup>.

### Ćwiczenie 2.55

Ewa Lu Ator wprowadziła do interpretera wyrażenie

```
(car 'abrakadabra)
```

Ku jej zaskoczeniu interpreter wypisał `quote`. Wyjaśnij to.

### 2.3.2. Przykład: różniczkowanie symboliczne

W celu zilustrowania operowania symbolami i dalszego zilustrowania abstrakcji danych rozważmy konstrukcję procedury wykonującej symboliczne różniczkowanie wyrażeń algebraicznych. Chcielibyśmy, aby argumentami procedury były wyrażenie algebraiczne i zmienna, a wynikiem pochodna wyrażenia względem tej zmiennej. Jeśli na przykład argumentami procedury są  $ax^2+bx+c$  i  $x$ , to wynikiem procedury powinno być  $2ax+b$ . Różniczkowanie symboliczne ma szczególne znaczenie w historii Lispu. Było ono jednym z przykładów motywujących do opracowania języka programowania przeznaczonego do operowania na symbolach. Ponadto wyznaczyło ono początek kierunku badań, które doprowadziły do opracowania potężnych systemów wykonujących symboliczne działania matematyczne, używanych aktualnie przez coraz większe grono matematyków stosowanych i fizyków.

Pracującając program różniczkujący symbolicznie, będziemy postępowali zgodnie z tą samą strategią abstrakcji danych, zgodnie z którą w punkcie 2.1.1 opracowaliśmy system liczb wymiernych. Oznacza to, że najpierw zdefiniujemy algorytm różniczkowania operujący na obiektach abstrakcyjnych, takich jak „sumy”, „iloczyny” czy „zmienne”, nie dbając o ich reprezentację, a dopiero potem zajmiemy się problemem ich reprezentacji.

### Program różniczkujący operujący na danych abstrakcyjnych

Dla prostoty będziemy rozważać bardzo prosty program różniczkowania symbolicznego, działający dla wyrażeń zbudowanych tylko za pomocą dwuargumentowego dodawania i mnożenia. Pochodną dowolnego takiego wyrażenia można wyznaczyć, stosując następujące upraszczające reguły:

<sup>36</sup> W praktyce programiści używają predykatu `equal?` do porównywania list, które zawierają zarówno liczby, jak i symbole. Liczby nie są uważane za symbole. Kwestia, czy dwie numerycznie równe liczby (zgodnie z `=`) spełniają również predykat `eq?`, w dużym stopniu zależy od implementacji. Lepsza definicja predykatu `equal?` (np. taka jak definicja predykatu pierwotnego w języku Scheme) powinna również określać, że dwie liczby `a` i `b` spełniają predykat `equal?`, jeżeli są numerycznie równe.

$$\frac{dc}{dx} = 0, \quad \text{jeśli } c \text{ jest stałą lub zmienną różną od } x$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$$

Zwróćmy uwagę, że ostatnie dwie reguły mają charakter rekurencyjny. Oznacza to, że aby otrzymać pochodną sumy, musimy najpierw wyznaczyć pochodne składników, a następnie dodać je do siebie. Każdy ze składników może być z kolei wyrażeniem wymagającym dalszego rozbicia. Rozbijanie na coraz to mniejsze kawałki prowadzi w końcu do otrzymywania kawałków, które są albo stałymi, albo zmiennymi o pochodnych równych 0 lub 1.

Aby zatrzymać te reguły w procedurze, pozwolimy sobie na odrobinę pobożnych życzeń, jak to już miało miejsce w trakcie konstruowania implementacji liczb wymiernych. Gdybyśmy dysponowali reprezentacją wyrażeń algebraicznych, to powinniśmy móc określić, czy dane wyrażenie jest sumą, iloczynem, stałą lub zmienną. Powinniśmy też móc wydobyć składowe tego wyrażenia. W przypadku sumy na przykład chcielibyśmy móc wydobyć obydwa składniki. Powinniśmy również móc budować wyrażenia ze składowych. Założymy, że dysponujemy procedurami implementującymi następujące selektory, konstruktory i predykaty:

<code>(variable? e)</code>	sprawdza, czy <code>e</code> jest zmienną;
<code>(same-variable? v1 v2)</code>	sprawdza, czy <code>v1</code> i <code>v2</code> są tą samą zmienną;
<code>(sum? e)</code>	sprawdza, czy <code>e</code> jest sumą;
<code>(addend e)</code>	pierwszy składnik sumy <code>e</code> ;
<code>(augend e)</code>	drugi składnik sumy <code>e</code> ;
<code>(make-sum a1 a2)</code>	tworzy sumę <code>a1</code> i <code>a2</code> ;
<code>(product? e)</code>	sprawdza, czy <code>e</code> jest iloczynem;
<code>(multiplier e)</code>	mnożnik iloczynu <code>e</code> ;
<code>(multiplicand e)</code>	mnożna iloczynu <code>e</code> ;
<code>(make-product m1 m2)</code>	tworzy iloczyn <code>m1</code> i <code>m2</code> .

Za ich pomocą oraz za pomocą predykatu pierwotnego `number?`, rozpoznającego liczby, możemy wyrazić reguły różniczkowania w postaci następującej procedury:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-product (multiplier exp)
                       (deriv (multiplicand exp) var)))
        ((make-product (deriv (multiplier exp) var)
                       (multiplicand exp))))
        (else
         (error "Nieznanego rodzaju wyrażenie -- DERIV" exp))))
```

Ta procedura `deriv` zawiera pełny algorytm różniczkowania. Dzięki temu, że jest ona wyrażona za pomocą danych abstrakcyjnych, będzie działać bez względu na to, jaką reprezentację wyrażeń algebraicznych wybierzemy, o ile tylko stworzymy odpowiedni zestaw selektorów i konstruktorów. Tą kwestią musimy zająć się jako następną.

### Reprezentowanie wyrażeń algebraicznych

Można sobie wyobrazić wiele sposobów użycia struktur listowych do reprezentowania wyrażeń algebraicznych. Moglibyśmy na przykład użyć list symboli odzwierciedlających tradycyjny zapis algebraiczny, reprezentując  $ax + b$  w postaci listy `(a*x + b)`. Jednakże szczególnie prosty sposób polega na zastosowaniu takiej samej notacji prefiksowo-nawiasowej, jaką jest stosowana w Lispie do zapisu kombinacji; tzn.  $ax + b$  reprezentujemy jako `(+ (* a x) b)`. Nasza reprezentacja danych na potrzeby problemu różniczkowania wygląda wówczas następująco:

- Zmienne to symbole. Rozpoznajemy je za pomocą predykatu pierwotnego `symbol?`:

```
(define (variable? x) (symbol? x))
```

- Dwie zmienne są takie same, jeśli reprezentujące je symbole spełniają `eq?`:

```
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
```

- Sumy i iloczyny są zbudowane jako listy:

```
(define (make-sum a1 a2) (list '+ a1 a2))
```

```
(define (make-product m1 m2) (list '* m1 m2))
```

- Suma jest listą, której pierwszym elementem jest symbol +:

```
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
```

- Pierwszy składnik jest drugim elementem listy tworzącej sumę:

```
(define (addend s) (cadr s))
```

- Drugi składnik jest trzecim elementem listy tworzącej sumę:

```
(define (augend s) (caddr s))
```

- Iloczyn jest listą, której pierwszym elementem jest symbol \*:

```
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
```

- Mnożnik jest drugim elementem listy tworzącej iloczyn:

```
(define (multiplier p) (cadr p))
```

- Mnożna jest trzecim elementem listy tworzącej iloczyn:

```
(define (multiplicand p) (caddr p))
```

Tak więc wystarczy połączyć powyższe procedury z algorytmem zawartym w procedurze deriv, a otrzymamy działający program różniczkowania symbolicznego. Przyjrzyjmy się kilku przykładom jego działania:

```
(deriv '(+ x 3) 'x)
(+ 1 0)

(deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))

(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
  (* (+ (* x 0) (* 1 y))
      (+ x 3)))
```

Wyniki, które daje program, są poprawne; jednakże nie są one uproszczone. To prawda, że

$$\frac{d(xy)}{dx} = x \cdot 0 + 1 \cdot y$$

ale wolelibyśmy, aby program wiedział, że  $x \cdot 0 = 0$ ,  $1 \cdot y = y$  oraz  $0 + y = y$ . Wynik dla drugiego przykładu powinien być po prostu równy  $y$ . Jak pokazuje trzeci przykład, gdy wyrażenia są złożone, staje się to poważnym problemem.

Trudność ta przypomina problem, jaki napotkaliśmy przy implementacji liczb wymiernych — nie uprościliśmy wyników. Aby zrealizować upraszczanie liczb wymiernych, wystarczyło tylko zmienić w implementacji konstruktory i selektory. Tutaj możemy zastosować podobną strategię. W ogóle nie będziemy zmieniać procedury `deriv`. Zamiast tego tak zmienimy procedurę `make-sum`, aby w przypadku, gdy oba składniki są liczbami, `make-sum` dodawała je i dawała w wyniku ich sumę. Również w przypadku, gdy jeden ze składników jest równy 0, `make-sum` powinna dawać w wyniku drugi składnik.

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
```

Korzystamy tutaj z procedury `=number?`, która sprawdza, czy dane wyrażenie jest równe danej liczbie:

```
(define (=number? exp num)
  (and (number? exp) (= exp num)))
```

Podobnie zmieniamy procedurę `make-product`, wbudowując w nią reguły mówiące, że 0 razy cokolwiek jest równe 0 oraz cokolwiek razy 1 jest równe samemu sobie:

```
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

Oto jak ta wersja działa dla naszych trzech przykładów:

```
(deriv '(+ x 3) 'x)
1
(deriv '(* x y) 'x)
y
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

Chociaż nastąpiła znaczna poprawa, trzeci przykład pokazuje, że zostało jeszcze dużo do zrobienia, zanim otrzymamy program, który zapisuje wyrażenia

w postaci, co do której moglibyśmy się zgodzić, że jest „najprostsza”. Problem upraszczania wyrażeń algebraicznych jest złożony z wielu powodów; między innymi postać, która może być najprostsza do jednego celu, może nie być najprostsza do innego celu.

### Ćwiczenie 2.56

Pokaż, jak rozszerzyć podstawowy program różniczkujący, aby radził sobie z większą liczbą rodzajów wyrażeń. Zaimplementuj na przykład regułę różniczkowania

$$\frac{d(u^n)}{dx} = nu^{n-1} \left( \frac{du}{dx} \right)$$

dodając nową klauzulę w programie `deriv` i definiując odpowiednie procedury: `exponentiation?` (predykat sprawdzający, czy wyrażenie jest potęgowaniem), `base` (podstawa), `exponent` (wykładnik) i `make-exponentiation` (konstruktor potęgowania). (Możesz użyć symbolu `**` na oznaczenie potęgowania). Wbuduj reguły mówiące, że cokolwiek podniesione do potęgi 0 jest równe 1 i cokolwiek podniesione do potęgi 1 jest równe samemu sobie.

### Ćwiczenie 2.57

Rozszerz program różniczkujący o możliwość operowania sumami i iloczynami dowolnej liczby (dwóch lub więcej) podwyrażeń. Ostatni przykład z tego punktu można zapisać jako

```
(deriv '(* x y (+ x 3)) 'x)
```

Spróbuj to zrobić, zmieniając wyłącznie reprezentację sum i iloczynów, a nie zmieniając w ogóle procedury `deriv`. Na przykład `addend` sumy mógłby być pierwszym składnikiem, a `augend` mógłby być sumą pozostałych składników.

### Ćwiczenie 2.58

Przypuśćmy, że chcemy zmienić program różniczkujący tak, aby działał na zwykłej notacji matematycznej, w której `+` i `*` są operatorami infiksowymi, a nie prefiksowymi. Ponieważ program różniczkujący zdefiniowaliśmy za pomocą danych abstrakcyjnych, możemy zmodyfikować go tak, aby operował na różnych reprezentacjach wyrażeń, zmieniając jedynie predykaty, selektory i konstruktory definiujące reprezentację wyrażeń algebraicznych, na której program ma operować.

- Pokaż, jak tego dokonać, aby móc różniczkować wyrażenia algebraiczne przedstawione w postaci infiksowej, takie jak `(x + (3 * (x + (y + 2))))`. Dla uproszczenia zadania załóż, że `+` i `*` zawsze mają po dwa argumenty, a wyrażenia mają wszystkie nawiasy.
- Problem staje się znacznie trudniejszy, jeśli dopuścimy standardową notację algebraiczną, jak na przykład `(x + 3 * (x + y + 2))`, w której opuszczamy wszystkie zbędne nawiasy i zakładamy, że mnożenia są wykonywane przed dodawaniem. Czy potrafisz skonstruować odpowiednie predykaty, selektory i konstruktory dla takiej reprezentacji, tak aby nasz program różniczkujący nadal działał?

### 2.3.3. Przykład: reprezentowanie zbiorów

W poprzednich przykładach zbudowaliśmy reprezentacje dwóch rodzajów złożonych obiektów danych: liczb wymiernych i wyrażeń algebraicznych. W jednym z tych przykładów mogliśmy wybrać, czy chcemy upraszczać wyrażenia (skracać ułamki) w czasie ich konstruowania, czy też w czasie wywołania selektorów, lecz poza tym wybór reprezentacji tych struktur za pomocą list był prosty. Kiedy zajmiemy się reprezentacją zbiorów, okaże się, że wybór reprezentacji nie jest taki oczywisty. Faktycznie, istnieje wiele możliwych reprezentacji znaczaco różniących się od siebie pod wieloma względami.

Nieformalnie, zbiór jest po prostu kolekcją różnych obiektów. Aby podać bardziej precyzyjną definicję, zastosujemy metodę abstrakcji danych — tzn. zdefiniujemy „zbiór”, specyfikując operacje, których będziemy używać na zbiorach. Są to: `union-set`, `intersection-set`, `element-of-set?` i `adjoin-set`. Predykat `element-of-set?` określa, czy dany element należy do zbioru. Procedura `adjoin-set` ma dwa argumenty, obiekt i zbiór, a jej wynikiem jest zbiór zawierający elementy początkowego zbioru oraz dołączony dany element. Procedura `union-set` oblicza sumę dwóch zbiorów, czyli zbiór zawierający wszystkie takie elementy, które występują w którymkolwiek z argumentów. Procedura `intersection-set` oblicza część wspólną dwóch zbiorów, czyli zbiór zawierający tylko takie elementy, które występują w obydwu argumentach. Z punktu widzenia abstrakcji danych możemy stworzyć dowolną reprezentację implementującą powyższe operacje zgodnie z podaną interpretacją<sup>37</sup>.

#### Zbiory jako listy nieuporządkowane

Jednym ze sposobów reprezentacji zbioru jest lista jego elementów, na której żaden element nie występuje więcej niż raz. Zbiór pusty jest reprezentowany przez pustą listę. W takiej reprezentacji predykat `element-of-set?` przypo-

<sup>37</sup> Gdybyśmy chcieli być bardziej formalni, moglibyśmy określić, że „zgodnie z podaną interpretacją” oznacza, iż operacje spełniają zestaw reguł, takich jak te:

- Dla dowolnego zbioru  $S$  i dowolnego obiektu  $x$ :  $(\text{element-of-set? } x \text{ } (\text{adjoin-set } x \text{ } S))$  jest prawdziwe (nieformalnie: „W wyniku wstawienia elementu do zbioru powstaje zbiór zawierający ten element”).
- Dla dowolnych zbiorów  $S$  i  $T$  oraz dowolnego obiektu  $x$ :  $(\text{element-of-set? } x \text{ } (\text{union-set } S \text{ } T))$  jest równe  $(\text{or } (\text{element-of-set? } x \text{ } S) \text{ } (\text{element-of-set? } x \text{ } T))$  (nieformalnie: „Zbiór  $(\text{union } S \text{ } T)$  zawiera te elementy, które należą do  $S$  lub  $T$ ”).
- Dla każdego obiektu  $x$ :  $(\text{element-of-set? } x \text{ } '())$  jest fałszywe (nieformalnie: „Żaden obiekt nie należy do zbioru pustego”). [Autorzy zakładają tu, że zbiór pusty jest reprezentowany przez pustą listę. Wydaje się to przeczywać zasadzie abstrakcji danych. Zamiast tego można wprowadzić konstruktor zbioru pustego, np. o nazwie `empty-set`. Wówczas powyższa reguła miałaby postać

Dla każdego obiektu  $x$ :  $(\text{element-of-set? } x \text{ } \text{empty-set})$  jest fałszywe; przyp. tłum.].

mina procedurę `memq` z punktu 2.3.1. Używa ona `equal?` zamiast `eq?`, a co za tym idzie, elementy zbiorów nie muszą być symbolami:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set))))))
```

Korzystając z tej procedury, możemy napisać procedurę `adjoin-set`. Jeśli obiekt, który ma być wstawiony do zbioru, już do niego należy, to wynikiem jest po prostu ten zbiór. W przeciwnym razie za pomocą `cons` dodajemy obiekt do listy reprezentującej zbiór:

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

Dla procedury `intersection-set` możemy zastosować strategię rekurencyjną. Jeżeli wiemy, jak wyznaczyć część wspólną zbioru `set2` i `cdr` zbioru `set1`, to musimy jedynie rozstrzygnąć, czy do otrzymanego zbioru wstawić `car` zbioru `set1`. To jednak zależy od tego, czy `(car set1)` należy już do `set2`. W ten sposób otrzymujemy następującą procedurę:

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

Jedną z kwestii, które powinniśmy wziąć pod uwagę, tworząc reprezentację, jest wydajność. Rozważmy liczbę kroków, jakie muszą wykonać nasze operacje na zbiorach. Ponieważ wszystkie one korzystają z `element-of-set?`, szybkość tej operacji ma znaczący wpływ na wydajność implementacji zbiorów jako całości. Teraz w celu sprawdzenia, czy obiekt należy do zbioru, `element-of-set?` może wymagać przejrzenia całego zbioru. (W najgorszym przypadku okaże się, że obiekt nie należy do zbioru). Stąd, jeśli zbiór ma  $n$  elementów, `element-of-set?` może wykonać  $n$  kroków. Tak więc liczba wykonywanych kroków jest rzędu  $\Theta(n)$ . Liczba kroków wykonywanych przez procedurę `adjoin-set`, korzystającej z tej operacji, także jest rzędu  $\Theta(n)$ . W przypadku procedury `intersection-set`, która wywołuje predykat `element-of-set?` dla każdego elementu zbioru `set1`, liczba wykonywanych kroków jest iloczynem wielkości będących argumentami lub jest rzędu  $\Theta(n^2)$  w przypadku dwóch zbiorów  $n$ -elementowych. To samo odnosi się do `union-set`.

### Ćwiczenie 2.59

Zaimplementuj operację `union-set` dla reprezentacji zbiorów za pomocą list nieuporządkowanych.

### Ćwiczenie 2.60

Przyjęliśmy, że reprezentujemy zbiory jako listy bez powtórzeń. Przypuśćmy teraz, że dopuszczały powtórzenia. Na przykład zbiór  $\{1, 2, 3\}$  może być reprezentowany jako lista  $(2 \ 3 \ 2 \ 1 \ 3 \ 2 \ 2)$ . Skonstruj procedury `element-of-set?`, `adjoin-set`, `union-set` i `intersection-set`, które operują na takiej reprezentacji. Jak wydajność każdej z nich ma się do wydajności odpowiadających im procedur dla reprezentacji bez powtórzeń? Czy istnieją zastosowania, w których użyłbyś tej reprezentacji zamiast reprezentacji bez powtórzeń?

### Zbiory jako listy uporządkowane

Jednym ze sposobów przyspieszenia operacji na zbiorach jest taka zmiana reprezentacji, żeby elementy zbioru występowali na liście w porządku rosnącym. Jest nam do tego potrzebna jakaś metoda porównywania dwóch obiektów, abyśmy mogli określić, który z nich jest większy. Możemy na przykład porównywać symbole leksykograficznie lub możemy przyjąć pewną metodę nadawania obiektem nie powtarzających się numerów, a następnie porównywać elementy przez porównywanie odpowiadających im liczb. Aby uprościć nasze rozważania, ograniczymy się do przypadku, w którym elementami zbiorów są liczby, dzięki czemu będziemy mogli je porównywać za pomocą `>` i `<`. Zbiór liczb będziemy reprezentować, wymieniając jego elementy w kolejności rosnącej. Podczas gdy nasza poprzednia reprezentacja dopuszczała, żeby zbiór  $\{1, 3, 6, 10\}$  był reprezentowany poprzez wymienienie jego elementów w dowolnej kolejności, nasza nowa reprezentacja dopuszcza tylko listę  $(1 \ 3 \ 6 \ 10)$ .

Jedną z korzyści wypływających z uporządkowania możemy zaobserwować na przykładzie `element-of-set?` — sprawdzając, czy element należy do zbioru, nie musimy już przeglądać wszystkich jego elementów. Jeśli natkniemy się na element zbioru, który jest większy od poszukiwanego elementu, to wiemy, że element ten nie należy do zbioru:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

Ile kroków na tym oszczędzamy? W najgorszym przypadku poszukiwany element może być elementem największym w zbiorze, więc liczba kroków będzie taka sama jak dla reprezentacji nieuporządkowanej. Z kolei, jeśli szukamy elementów o wielu różnych wielkościach, to możemy się spodziewać, że czasami będziemy mogli kończyć przeszukiwanie w pobliżu początku listy, a kiedy

indziej będziemy musieli przeszukiwać większą część listy. Średnio możemy oczekwać przeszukiwania około połowy elementów w zbiorze. Tak więc średnia liczba wykonywanych kroków będzie wynosiła około  $n/2$ . Jest to wciąż rzędu  $\Theta(n)$ , ale średnio oszczędzamy połowę kroków w porównaniu z poprzednią implementacją.

Bardziej imponujące przyspieszenie uzyskujemy w przypadku procedury `intersection-set`. Przy reprezentacji nieuporządkowanej operacja ta wymagała  $\Theta(n^2)$  kroków, ponieważ dla każdego elementu ze zbioru `set1` przeszukiwaliśmy cały zbiór `set2`. Ale przy reprezentacji uporządkowanej możemy zastosować sprytniejszą metodę. Zaczynamy od porównania pierwszych elementów, `x1` i `x2`, dwóch danych zbiorów. Jeśli `x1` jest równe `x2`, to element ten należy do części wspólnej zbiorów, a pozostałe elementy części wspólnej tworzą część wspólną `cdr` pierwszego zbioru i `cdr` drugiego zbioru. Przypuśćmy jednak, że `x1` jest mniejsze niż `x2`. Ponieważ `x2` jest najmniejszym elementem w `set2`, możemy natychmiast stwierdzić, że `x1` nie należy do `set2`, a zatem nie należy do części wspólnej. Tak więc część wspólna zbiorów jest równa części wspólnej `set2` i `cdr` zbioru `set1`. Podobnie, jeśli `x2` jest mniejsze niż `x1`, to część wspólna zbiorów jest dana przez część wspólną `set1` i `cdr` zbioru `set2`. Oto procedura:

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1
                     (intersection-set (cdr set1)
                                       (cdr set2))))
               ((< x1 x2)
                (intersection-set (cdr set1) set2))
               ((< x2 x1)
                (intersection-set set1 (cdr set2)))))))
```

Aby określić liczbę wymaganych w tym procesie kroków, zauważmy, że z każdym krokiem sprowadzamy problem części wspólnej do obliczenia części wspólnej mniejszych zbiorów — usuwając pierwszy element z `set1` lub `set2`, lub z obydwu tych zbiorów. A zatem liczba wykonywanych kroków jest co najwyżej sumą wielkości `set1` i `set2`, a nie iloczynem tych wielkości, jak to miało miejsce przy reprezentacji nieuporządkowanej. Oznacza to rzad wielkości  $\Theta(n)$ , a nie  $\Theta(n^2)$  — znaczące przyspieszenie nawet dla stosunkowo niewielkich zbiorów.

### Ćwiczenie 2.61

Podaj implementację procedury `adjoin-set` korzystającej z reprezentacji uporządkowanej. Przez analogię do `element-of-set?` pokaż, jak skorzystać z uporządkowania elementów w celu uzyskania procedury wymagającej liczby kroków średnio o połowę mniejszej niż w przypadku reprezentacji nieuporządkowanej.

### Ćwiczenie 2.62

Podaj implementację procedury `union-set` o złożoności rzędu  $\Theta(n)$  dla reprezentacji zbiorów jako list uporządkowanych.

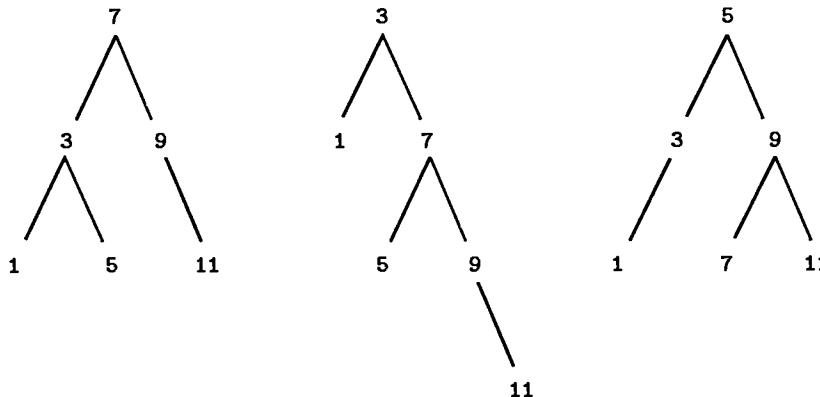
## Zbiory jako drzewa binarne

Możemy osiągnąć lepsze wyniki niż przy reprezentacji zbiorów za pomocą list uporządkowanych, organizując elementy zbioru w formie drzewa. Każdy węzeł drzewa zawiera jeden element zbioru, nazywany „wartością przechowywaną” w tym węźle, oraz dwa dowiązania do dwóch innych (być może pustych) węzłów. „Lewe” dowiązanie wskazuje na elementy mniejsze od elementu w danym węźle, a „prawe” dowiązanie na elementy większe. Na rysunku 2.16 są pokazane drzewa reprezentujące zbiór  $\{1, 3, 5, 7, 9, 11\}$ . Ten sam zbiór może być reprezentowany przez wiele różnych drzew. Jedyną rzeczą, jakiej wymagamy od prawidłowej reprezentacji, jest to, aby wszystkie elementy w lewym poddrzewie były mniejsze od wartości przechowywanej w węźle, a wszystkie elementy w prawym poddrzewie były od niej większe.

Korzyść wypływającą ze stosowania reprezentacji drzewiastej jest następująca. Przypuśćmy, że chcemy sprawdzić, czy liczba  $x$  należy do zbioru. Zaczynamy od porównania  $x$  z wartością przechowywaną w korzeniu (tzn. najwyższym węźle) drzewa. Jeśli  $x$  jest mniejsze od tej wartości, to wiemy, że wystarczy przeszukać tylko lewe poddrzewo; jeśli  $x$  jest większe, to wystarczy przeszukać tylko prawe poddrzewo. Teraz, jeśli drzewo jest „zrównoważone”, to każde z tych poddrzew będzie wielkości, mniej więcej, połowy całego drzewa. Tak więc w jednym kroku sprawdzamy problem przeszukania drzewa wielkości  $n$  do problemu przeszukania drzewa wielkości  $n/2$ . Ponieważ wielkość drzewa zmniejsza się z każdym krokiem o połowę, możemy oczekwać, że liczba kroków wykonanych w trakcie przeszukiwania drzewa wielkości  $n$  będzie rzędu  $\Theta(\log n)$ <sup>38</sup>. Dla dużych zbiorów jest to znaczące przyspieszenie w porównaniu z poprzednią reprezentacją.

Drzewa możemy reprezentować za pomocą list. Każdy węzeł będzie listą złożoną z trzech elementów: wartości przechowywanej w węźle, lewego poddrzewa i prawego poddrzewa. Jeśli lewe lub prawe poddrzewo będzie pustą li-

<sup>38</sup> Zmniejszanie rozmiaru problemu w każdym kroku o połowę jest cechą wyróżniającą algorytmy o złożoności logarytmicznej, tak jak to widzieliśmy na przykładach algorytmu szybkiego potęgowania w punkcie 1.2.4 i metody wyszukiwania binarnego w punkcie 1.3.3.



Rys. 2.16. Różne drzewa binarne reprezentujące zbiór {1, 3, 5, 7, 9, 11}

stą, oznaczać to będzie, że w danym miejscu nie jest dowiązane żadne poddrzewo. Reprezentację taką możemy opisać za pomocą następujących procedur<sup>39</sup>:

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
```

Teraz możemy zapisać procedurę `element-of-set?`, używając opisanej powyżej strategii:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Wstawianie elementu do drzewa można zaimplementować podobnie i wymaga to również  $\Theta(\log n)$  kroków. Aby wstawić element  $x$ , porównujemy go z wartością przechowywaną w węźle w celu określenia, czy należy go dodać do prawej, czy też lewej gałęzi. Po wstawieniu elementu do odpowiedniej gałęzi łączymy w jedną całość tę dopiero co skonstruowaną gałąź z wartością

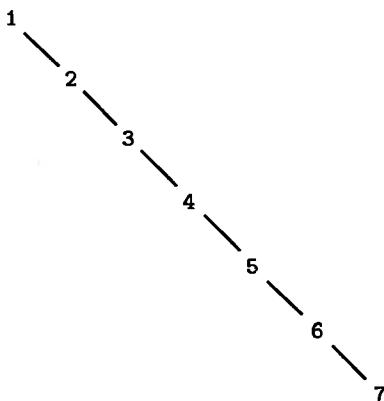
<sup>39</sup> Reprezentujemy zbiory za pomocą drzew, a drzewa za pomocą list — w rezultacie jedna abstrakcja danych korzysta z drugiej. Możemy traktować procedury `entry`, `left-branch`, `right-branch` i `make-tree` jako sposób odizolowania abstrakcji „drzew binarnych” od konkretnej reprezentacji listowej, jaką chcemy zastosować.

przechowywaną w węźle i drugą gałęzią. Jeśli  $x$  jest równe wartości przechowywanej w węźle, to wynikiem jest po prostu ten węzeł. Jeśli mamy wstawić  $x$  do drzewa pustego, to tworzymy drzewo, w którego korzeniu jest przechowywany  $x$  i którego poddrzewa lewe i prawe są puste. Oto procedura:

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set))))))
```

Powyzsze stwierdzenie, że przeszukiwanie drzewa jest wykonywane w logarytmicznej liczbie kroków opiera się na założeniu, że drzewo jest „zrównoważone”, tzn. lewe i prawe poddrzewa każdego węzła mają w przybliżeniu taką samą liczbę elementów, czyli każde poddrzewo zawiera około połowy elementów swojego rodzica. Jak jednak możemy być pewni, że tworzone przez nas drzewa będą zrównoważone? Nawet jeżeli zaczynamy od drzewa zrównoważonego, to dodając do niego elementy za pomocą `adjoin-set`, możemy otrzymać wynik niezrównoważony. Ponieważ położenie nowo wstawionego elementu zależy od wyników porównań tego elementu z elementami, które są już w zbiorze, więc możemy oczekiwąć, że jeżeli będziemy wstawiać elementy „losowo”, to przeciętnie drzewo będzie zwykle zrównoważone. Nie mamy jednak gwarancji, że tak będzie. Jeśli na przykład zaczniemy od zbioru pustego i wstawimy do niego po kolej liczb od 1 do 7, to uzyskamy drzewo silnie niezrównoważone, przedstawione na rys. 2.17. W drzewie tym wszystkie lewe poddrzewa są puste, a więc nie jest ono w niczym lepsze od zwykłej listy uporządkowanej. Jednym ze sposobów rozwiązania tego problemu jest zdefiniowanie operacji przekształcającej dowolne drzewo w drzewo zrównoważone zawierające te same elementy. Wówczas możemy wykonywać takie przekształcenie co kilka wywołań operacji `adjoin-set`, aby zrównoważyć nasz zbiór. Są również inne sposoby rozwiązania tego problemu; większość z nich wymaga opracowania nowej struktury danych, dla której operacje wyszukiwania i wstawiania mogą być wykonane w  $\Theta(\log n)$  krokach<sup>40</sup>.

<sup>40</sup> Wśród przykładów takich struktur można wymienić *B-drzewa* i *drzewa czerwono-czarne*. Literatura poświęcona strukturam danych związanym z tym problemem jest bogata — zob. [16].



Rys. 2.17. Drzewo niezrównoważone otrzymane w wyniku wstawienia kolejno liczb od 1 do 7

### Ćwiczenie 2.63

Obydwie poniższe procedury przekształcają drzewo binarne w listę.

```

(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1 (right-branch tree))))))

(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                     (cons (entry tree)
                           (copy-to-list (right-branch tree)
                                         result-list)))))

  (copy-to-list tree '()))
  
```

(a) Czy te dwie procedury dają takie same wyniki dla każdego drzewa? Jeśli nie, to czym się różnią wyniki? Jaki listy będą wynikami tych procedur dla drzew przedstawionych na rys. 2.16?

(b) Czy te dwie procedury wymagają liczby kroków takiego samego rzędu, aby przekształcić drzewo zrównoważone o  $n$  elementach w listę? Jeśli nie, to która ma mniejszą złożoność?

### Ćwiczenie 2.64

Następująca procedura `list->tree` przekształca listę uporządkowaną w zrównoważone drzewo binarne. Argumentami pomocniczej procedury `partial-tree` są liczba całkowita  $n$  i co najmniej  $n$ -elementowa lista. Tworzy ona zrównoważone drzewo binarne zawierające pierwszych  $n$  elementów listy. Wynikiem `partial-tree` jest para

(zbudowana za pomocą `cons`), której `car` to utworzone drzewo, a `cdr` to lista elementów, których nie umieszczono w drzewie.

```
(define (list->tree elements)
  (car (partial-tree elements (length elements)))))

(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1)))))
            (let ((this-entry (car non-left-elts))
                  (right-result (partial-tree (cdr non-left-elts)
                                              right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts (cdr right-result)))
                (cons (make-tree this-entry left-tree right-tree)
                      remaining-elts)))))))
```

- (a) Napisz, tak jasno jak potrafisz, krótki akapit wyjaśniający, jak działa `partial-tree`. Narysuj drzewo tworzone przez `list->tree` dla listy `(1 3 5 7 9 11)`.
- (b) Jakiego rzędu jest liczba kroków wykonywanych przez `list->tree` przy przekształcaniu  $n$ -elementowej listy?

### Ćwiczenie 2.65

Zastosuj wyniki ćwiczeń 2.63 i 2.64 do zaimplementowania operacji `union-set` i `intersection-set` o złożoności  $\Theta(n)$  dla zbiorów reprezentowanych przez (zrównoważone) drzewa binarne<sup>41</sup>.

### Zbiory i wyszukiwanie informacji

Zbadaliśmy możliwość zastosowania list do reprezentowania zbiorów i zobaczyliśmy, jak duży wpływ może mieć wybór reprezentacji obiektów danych na efektywność programów korzystających z tych danych. Inna przyczyna skoncentrowania się na zbiorach jest taka, że omawiane tutaj techniki pojawiają się wielokrotnie w zastosowaniach związanych z wyszukiwaniem informacji.

Rozważmy bazę danych zawierającą dużą liczbę pojedynczych rekordów, taką jak akta osobowe w przedsiębiorstwie bądź transakcje w systemie rozliczeniowym. Typowy system zarządzania danymi poświęca dużą część czasu na uzyskiwanie dostępu lub modyfikowanie danych w rekordach i dlatego wymaga efektywnych metod uzyskiwania dostępu do rekordów. Jest to realizowane przez wyznaczenie części każdego rekordu jako jego *klucza* identyfikacyjnego.

<sup>41</sup> Ćwiczenia 2.63–2.65 pochodzą od Paula Hilfingera.

Klucz może być czymkolwiek, co jednoznacznie wyznacza rekord. W przypadku danych osobowych może to być numer identyfikacyjny pracownika. W przypadku systemu rozliczeniowego może to być numer transakcji. Czymkolwiek jest klucz, gdy definiujemy rekord jako strukturę danych, powinniśmy utworzyć selektor key, którego wartością jest klucz związany z danym rekordem.

Możemy teraz reprezentować bazę danych jako zbiór rekordów. W celu odszukania rekordu o zadanym kluczu używamy procedury `lookup`, której argumentami są klucz i baza danych, a wynikiem jest rekord o danej wartości klucza lub wartość fałsz, jeśli takiego rekordu nie ma. Procedura `lookup` jest zaimplementowana prawie tak samo jak `element-of-set?`. Jeśli na przykład zbiór rekordów jest zaimplementowany jako nieuporządkowana lista rekordów, to możemy zastosować

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) false)
        ((equal? given-key (key (car set-of-records)))
         (car set-of-records))
        (else (lookup given-key (cdr set-of-records))))))
```

Oczywiście są lepsze sposoby reprezentowania dużych zbiorów niż w postaci list nieuporządkowanych. Systemy wyszukiwania informacji, w których rekordy mają być „dostępne bezpośrednio”, są zwykle zaimplementowane za pomocą metod opartych na drzewach, takich jak omówiona wcześniej reprezentacja za pomocą drzew binarnych. Technika abstrakcji danych może być bardzo pomocna przy konstruowaniu takiego systemu. Programista może stworzyć początkową implementację, stosując łatwe i proste reprezentacje, takie jak listy nieuporządkowane. Są one nieodpowiednie dla ostatecznego systemu, ale mogą być przydatne jako „prototyp” bazy danych, którego można użyć do testowania pozostały części systemu. Później reprezentacja danych może być zmieniona na bardziej skomplikowaną. Jeśli dostęp do bazy danych jest realizowany poprzez abstrakcyjne selektory i konstruktory, to taka zmiana reprezentacji nie wymaga żadnych zmian w pozostały części systemu.

### Ćwiczenie 2.66

Zaimplementuj procedurę `lookup` dla wariantu, gdy zbiór rekordów tworzy drzewo binarne uporządkowane zgodnie z wartościami liczbowymi kluczy.

#### 2.3.4. Przykład: drzewa kodów Huffmmana

W niniejszym punkcie będziemy ćwiczyć użycie struktur listowych i abstrakcji danych w celu operowania na zbiorach i drzewach. Zastosowanie to odnosi się do metod reprezentowania danych w postaci ciągów zer i jedynek (bitów). Na przykład w kodzie ASCII, używanym do reprezentowania tekstów w komputerach, każdy znak jest przedstawiany jako ciąg siedmiu bitów. Używając siedmiu

bitów, możemy rozróżnić  $2^7$ , czyli 128, różnych znaków. Ogólnie mówiąc, jeśli chcemy przedstawić  $n$  różnych symboli, będziemy potrzebowali  $\log_2 n$  bitów na symbol. Jeśli wszystkie nasze wiadomości składają się z ośmiu symboli: A, B, C, D, E, F, G i H, to możemy określić ich kodowanie przy użyciu trzech bitów na znak; na przykład:

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

Przy takim kodowaniu wiadomość

BACADAEAFABBAAGAH

jest zakodowana jako ciąg 54 bitów:

00100001000001100010000010100000100100000000110000111

Takie kody, jak ASCII i powyższy kod „od-A-do-H”, są znane jako kody o *stałej długości*, ponieważ każdy symbol jest w nich reprezentowany za pomocą takiej samej liczby bitów. Czasami bardziej korzystne jest zastosowanie kodów o *zmiennej długości*, w których różne symbole mogą być reprezentowane za pomocą różnej liczby bitów. W kodzie Morse'a na przykład różne litery alfabetu są reprezentowane za pomocą różnej liczby kresek i kropek. W szczególności, najczęściej występująca litera E jest reprezentowana przez pojedynczą kropkę. Na ogół, jeśli w naszych wiadomościach pewne symbole pojawiają się bardzo często, a inne bardzo rzadko, to możemy dane kodować efektywniej (tzn. używając mniejszej liczby bitów na wiadomość), przyporządkowując krótsze kody częściej pojawiającym się symbolom. Rozważmy następujący alternatywny kod dla liter od A do H:

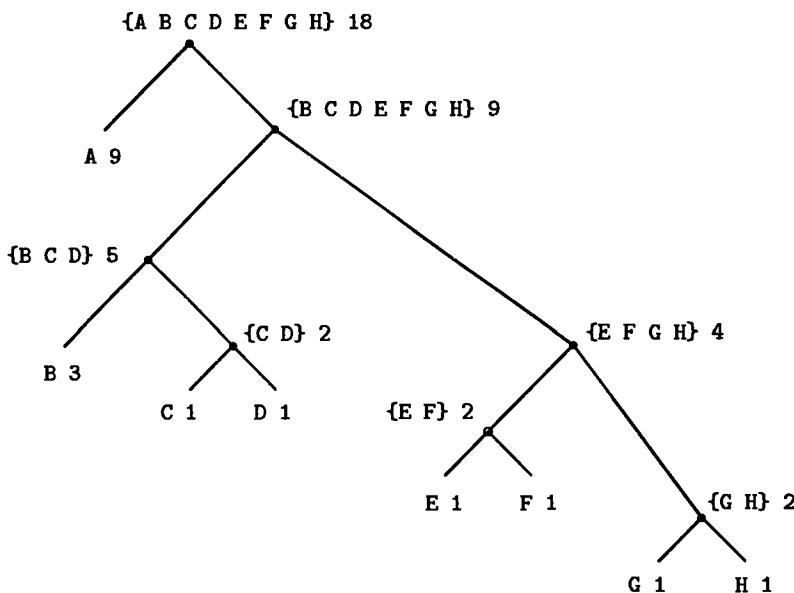
A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

Przy takim kodowaniu ta sama wiadomość co wcześniej jest zakodowana jako

100010100101101100011010100100000111001111

Ten ciąg składa się z 42 bitów, a więc w porównaniu z kodem o stałej długości przedstawionym powyżej mamy tu oszczędność 20% pamięci.

Jedną z trudności przy używaniu kodów o zmiennej długości jest określenie, kiedy, czytając sekwencję zer i jedynek, dochodzimy do końca symbolu. W przypadku kodu Morse'a rozwiązano ten problem, stosując po ciągu kropki i kresek reprezentujących każdą literę specjalny *kod rozdzielający* (w tym przypadku pauzę). Inne rozwiązanie polega na opracowaniu kodu w taki sposób, aby pełny kod żadnego symbolu nie był początkowym fragmentem (prefiksem) kodu innego symbolu. Taki kod jest nazywany *kodem prefiksowym*. W powyż-



Rys. 2.18. Drzewo kodu Huffmana

szym przykładzie A jest kodowane przez 0, a B przez 100, więc żaden inny symbol nie może mieć kodu rozpoczynającego się od 0 lub 100.

Na ogół możemy uzyskać znaczne oszczędności, jeżeli będziemy używać kodów prefiksowych o zmiennej długości, w których bierze się pod uwagę względne częstotliwości występowania symboli w kodowanych wiadomościach. Mетодa kodowania Huffmana, nazwana tak na cześć jej twórcy Davida Huffmana, jest schematem tworzenia takich kodów. Kod Huffmana może być przedstawiony jako drzewo binarne, którego liście są kodowanymi symbolami. W każdym nie będącym liściem węzle drzewa znajduje się zbiór wszystkich symboli przechowywanych w liściach leżących poniżej danego węzła. Dodatkowo każdy symbol znajdujący się w liściu ma przypisaną wagę (która jest jego względową częstotliwością występowania), a każdy węzeł nie będący liściem ma przypisaną wagę będącą sumą wag wszystkich liści leżących poniżej niego. Wagi te nie są używane ani w procesie kodowania, ani dekodowania. Dalej zobaczymy, w jaki sposób używa się ich przy tworzeniu drzewa.

Na rysunku 2.18 jest przedstawione drzewo Huffmana dla omówionego wcześniej kodu „od-A-do-H”. Wagi liści wskazują na to, że drzewo zostało zaprojektowane dla wiadomości, w których A pojawia się ze względną częstotliwością 9, B ze względną częstotliwością 3, a pozostałe litery ze względną częstotliwością 1.

Mając drzewo Huffmana, możemy wyznaczyć kod dowolnego symbolu, poruszając się w dół od korzenia do liścia zawierającego dany symbol. Za każdym razem, gdy przechodzimy w dół wzdłuż lewej gałęzi, dodajemy do kodu 0,

a gdy przechodzimy w dół wzduż prawej gałęzi, dodajemy do kodu 1. (Wybieramy zawsze taką gałąź, która prowadzi do liścia zawierającego dany symbol lub do węzła zawierającego zbiór, do którego należy dany symbol). Na przykład, zaczynając od korzenia drzewa pokazanego na rys. 2.18, dochodzimy do liścia oznaczającego literę D, wybierając najpierw prawą gałąź, potem lewą, a następnie dwukrotnie prawą; stąd kodem D jest 1011.

Chcąc zdekodować ciąg bitów za pomocą drzewa Huffmana, zaczynamy przemieszczać się od korzenia drzewa, wybierając lewe lub prawe gałęzie zgodnie z kolejnymi zerami i jedynkami w ciągu bitów. Za każdym razem, gdy dochodzimy do liścia, otrzymujemy kolejny symbol wiadomości; po czym zaczynamy znowu od korzenia drzewa, dekodując następny symbol. Założymy na przykład, że mamy dane powyższe drzewo i ciąg 10001010. Zaczynając od korzenia, przechodzimy w dół wzduż prawej gałęzi (gdyż pierwszy bit ciągu jest równy 1), następnie wzduż lewej gałęzi (gdyż drugi bit ciągu jest równy 0), a potem wzduż lewej gałęzi (gdyż trzeci bit ciągu jest także równy 0). W ten sposób dochodzimy do liścia odpowiadającego literze B, a więc pierwszą literą dekodowanej wiadomości jest B. Zaczynamy ponownie od korzenia i przechodzimy w lewo, ponieważ kolejny bit w ciągu jest równy 0. Dochodzimy do liścia reprezentującego literę A. Następnie znowu zaczynamy od korzenia i zgodnie z pozostałym ciągiem bitów 1010 idziemy w prawo, w lewo, w prawo i w lewo, dochodząc do C. Tak więc cała wiadomość to BAC.

### Tworzenie drzew Huffmmana

Jak zbudować „najlepszy” kod, mając dany „alfabet” i względne częstotliwości występowania symboli? (Inaczej mówiąc, jakie drzewo koduje wiadomości w najmniejszej liczbie bitów?) Huffman podał algorytm tworzenia takiego drzewa i pokazał, że powstający kod jest faktycznie najlepszym kodem prefiksowym o zmiennej długości dla wiadomości, w których względne częstotliwości występowania symboli odpowiadają względnym częstotliwościami, dla których drzewo utworzono. Nie będziemy tutaj dowodzić optymalności kodów Huffmmana [zobacz [16], p. 17.3; przyp. tłum.], ale pokażemy jak konstruować drzewa Huffmmana<sup>42</sup>.

Algorytm konstruowania drzewa Huffmmana jest bardzo prosty. Pomysł polega na takim ułożeniu węzłów, aby symbole o najmniejszych częstotliwościach znajdowały się najdalej od korzenia. Najpierw tworzymy zbiór liści zawierających symbole i ich częstotliwości, zgodnie z danymi źródłowymi, na podstawie których budujemy kod. Następnie znajdujemy dwa liście o najmniejszych wagach i łączymy je, tworząc węzeł, którego lewa i prawa gałąź prowadzą do tych liści. Waga tego nowego węzła jest sumą wag złączonych liści. Usuwamy te dwa liście ze zbioru początkowego i zastępujemy je nowym węzłem. Kontynuujemy ten proces. W każdym kroku łączymy dwa węzły o najmniej-

<sup>42</sup> Omówienie matematycznych własności kodów Huffmmana można znaleźć w [40].

szycz wagach, usuwamy je ze zbioru i zastępujemy węzłem, do którego są one dowiązane poprzez lewą i prawą gałąź. Przerywamy proces, gdy mamy tylko jeden węzeł, który jest korzeniem całego drzewa. Oto jak powstało drzewo Huffmana z rys. 2.18:

Początkowe liście	{(A 9) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}
Złączenie	{(A 9) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}
Złączenie	{(A 9) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}
Złączenie	{(A 9) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}
Złączenie	{(A 9) (B 3) ({C D} 2) ({E F G H} 4)}
Złączenie	{(A 9) ({B C D} 5) ({E F G H} 4)}
Złączenie	{(A 9) ({B C D E F G H} 9)}
Ostatnie złączenie	(({A B C D E F G H} 18)}

Algorytm ten nie zawsze jednoznacznie wyznacza drzewo, ponieważ w każdym kroku może być wiele węzłów o najmniejszych wagach. Również porządek, w jakim łączymy węzły (tzn. który jest po lewej, a który po prawej stronie), jest dowolny.

### Reprezentowanie drzew Huffmana

W poniższych ćwiczeniach będziemy pracować nad systemem używającym drzew Huffmana do kodowania i dekodowania wiadomości oraz tworzącym drzewa Huffmana zgodnie z naszkicowanym powyżej algorytmem. Zaczniemy od omówienia, jak drzewa są reprezentowane.

Liście drzewa są reprezentowane przez listy zawierające symbol `leaf`, symbol przechowywany w liściu oraz wagę:

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (leaf? object)
  (eq? (car object) 'leaf))

(define (symbol-leaf x) (cadr x))
(define (weight-leaf x) (caddr x))
```

Ogólne drzewo jest listą złożoną z lewej gałęzi, prawej gałęzi, zbioru symboli i wagi. Zbiór symboli jest po prostu listą symboli, a nie jakąś bardziej wyrafinowaną reprezentacją. Gdy tworzymy drzewo, łącząc dwa węzły, wagę tego drzewa otrzymujemy jako sumę wag węzłów, a zbiór symboli jako sumę zbiorów symboli węzłów. Ponieważ zbiorów symboli są reprezentowane jako listy, możemy je więc sumować za pomocą procedury `append` zdefiniowanej w punkcie 2.2.1:

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))
```

Jeśli tworzymy drzewo w opisany sposób, to mamy następujące selektory:

```
(define (left-branch tree) (car tree))

(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))

(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (caddar tree)))
```

Procedury `symbols` i `weight` muszą działać trochę inaczej — w zależności od tego, czy są wywoływanie dla liścia, czy też dla ogólnego drzewa. Są to proste przykłady *procedur ogólnych* (procedur, które mogą działać dla więcej niż jednego rodzaju danych), o których będziemy mieli dużo więcej do powiedzenia w podrozdziałach 2.4 i 2.5.

### Procedura dekodująca

Następująca procedura implementuje algorytm dekodujący. Jej argumentami są lista zer i jedynek oraz drzewo Huffmana.

```
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
              (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch))))))
  (decode-1 bits tree))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "Zła wartość bitu -- CHOOSE-BRANCH" bit))))
```

Procedura `decode-1` ma dwa argumenty: listę pozostałych bitów i pozycję bieżącą w drzewie. Cały czas porusza się „w dół” drzewa, wybierając lewą lub prawą gałąź w zależności od tego, czy kolejny bit na liście jest równy zero czy jeden. (Dzieje się to z użyciem procedury `choose-branch`). Gdy dochodzimy do liścia, kolejnym symbolem wiadomości jest symbol przechowywany w tym liściu. Dołączamy go, za pomocą `cons`, do reszty wiadomości, którą dekodujemy, zaczynając od korzenia drzewa. Zwróćmy uwagę na kontrolę błędów w ostatniej klauzuli `choose-branch`, która wypisuje komunikat w przypadku napotkania w danych wejściowych czegokolwiek innego niż zero lub jeden.

## Zbiory elementów ważonych

W naszej reprezentacji drzew każdy węzeł nie będący liściem zawiera zbiór symboli przedstawionych w postaci zwykłej listy. Jednakże omówiony powyżej algorytm tworzący drzewa wymaga również operacji na zbiorach liści i drzew, łącząc kolejno dwa najmniejsze elementy. Ponieważ będziemy musieli wielokrotnie znajdować najmniejszy element w zbiorze, wygodnie byłoby użyć uporządkowanej reprezentacji takiego rodzaju zbiorów.

Zbiór liści i drzew będziemy reprezentować jako listę elementów, uporządkowaną zgodnie z rosnącymi wagami elementów. Następujący konstruktor `adjoin-set` jest podobny do opisanego w ćwiczeniu 2.61; jednakże porównywane są wagi elementów, a element wstawiany do zbioru wcześniej do niego nie należy.

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                     (adjoin-set x (cdr set))))))
```

Następująca procedura na podstawie listy par symbol–częstość, takiej jak ((A 4) (B 2) (C 1) (D 1)), tworzy początkowy uporządkowany zbiór list, gotowy do łączenia elementów zgodnie z algorytmem Huffmana:

```
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair)      ; symbol
                               (cadr pair)) ; częstość
                    (make-leaf-set (cdr pairs)))))))
```

**Ćwiczenie 2.67**

Zdefiniuj drzewo kodu i próbna wiadomość:

```
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
    (make-code-tree
      (make-leaf 'B 2)
      (make-code-tree (make-leaf 'D 1)
        (make-leaf 'C 1)))))

(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 0))
```

Użyj procedury `decode` do zdekodowania wiadomości i podaj wynik.

**Ćwiczenie 2.68**

Argumentami procedury `encode` są wiadomość i drzewo, a jej wynikiem jest lista bitów stanowiących zakodowaną wiadomość.

```
(define (encode message tree)
  (if (null? message)
    '()
    (append (encode-symbol (car message) tree)
      (encode (cdr message) tree))))
```

`Encode-symbol` to procedura, którą musisz napisać. Jej wynikiem jest lista bitów kodujących dany symbol zgodnie z danym drzewem. Powinieneś skonstruować `encode-symbol` tak, aby zgłaszała błąd w przypadku, gdy dany symbol w ogóle nie występuje w drzewie. Przetestuj swoją procedurę, kodując wynik otrzymany w ćwiczeniu 2.67 za pomocą próbnego drzewa i sprawdzając, czy uzyskany wynik jest taki sam jak próbna wiadomość.

**Ćwiczenie 2.69**

Argumentem poniższej procedury jest lista par symbol-częstość (gdzie żaden symbol nie występuje więcej niż w jednej parze). Procedura ta tworzy drzewo kodu zgodnie z algorytmem Huffmmana.

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))
```

`Make-leaf-set` to opisana powyżej procedura przekształcająca listę par w uporządkowany zbiór liści. `Successive-merge` to procedura, którą musisz napisać, używając `make-code-tree` do łączenia kolejno elementów zbioru o najmniejszych wagach aż do uzyskania tylko jednego elementu w zbiorze, który jest poszukiwanym drzewem Huffmmana. (Procedura ta może sprawić trochę trudności, ale nie jest naprawdę skomplikowana. Jeśli okaże się, że konstrujesz procedurę złożoną, to prawie na pewno coś robisz źle. Możesz w znaczący sposób skorzystać z faktu, że używamy uporządkowanej reprezentacji zbiorów).

### Ćwiczenie 2.70

Następujący ośmiosymbolowy alfabet wraz ze związanymi z nim względnymi częstotliwościami skonstruowano w celu efektywnego kodowania tekstów piosenek rockowych lat pięćdziesiątych XX w. (Zwróć uwagę, że „symbolami” „alfabetu” nie muszą być pojedyncze litery).

A	2	NA	16
BOOM	1	SHA	3
GET	2	YIP	9
JOB	2	WAH	1

Użyj procedury `generate-huffman-tree` (z ćwiczenia 2.69) do utworzenia odpowiedniego drzewa Huffmana oraz procedury `encode` (z ćwiczenia 2.68) do zakodowania następującej wiadomości:

Get a job  
Sha na na na na na na na  
Get a job  
Sha na na na na na na na  
Wah yip yip yip yip yip yip yip  
Sha boom

Ille bitów potrzeba do jej zakodowania? Jaka jest najmniejsza liczba bitów potrzebnych do zakodowania tej piosenki przy użyciu kodu o stałej długości dla ośmiosymbolowego alfabetu?

### Ćwiczenie 2.71

Przypuśćmy, że mamy drzewo Huffmana dla alfabetu o  $n$  symbolach i że względne częstotliwości symboli wynoszą  $1, 2, 4, \dots, 2^{n-1}$ . Narysuj drzewo dla  $n = 5$  i  $n = 10$ . Ille bitów potrzeba w takim drzewie (dla dowolnego  $n$ ) do zakodowania najczęściej pojawiającego się symbolu? A ile dla najrzadziej pojawiającego się symbolu?

### Ćwiczenie 2.72

Rozważmy procedurę kodującą skonstruowaną w ćwiczeniu 2.68. Jakiego rzędu jest liczba kroków potrzebnych do zakodowania symbolu? Nie zapomnij uwzględnić tu liczby kroków potrzebnych do przeszukania listy symboli w każdym napotkanym węźle. Odpowiedź na to pytanie jest w ogólności trudna. Rozważ przypadek szczególny, gdy względne częstotliwości  $n$  symboli są takie jak opisane w ćwiczeniu 2.71, i podaj rzęd wielkości (w zależności od  $n$ ) liczby kroków potrzebnych do zakodowania najczęściej i najrzadziej pojawiających się symboli z alfabetu.

## 2.4. Wielorakie reprezentacje danych abstrakcyjnych

Wprowadziliśmy zasadę abstrakcji danych jako takiej techniki konstruowania systemów, dzięki której duża część programu może zostać określona niezależnie od decyzji dotyczących implementacji obiektów danych, którymi operuje program. W punkcie 2.1.1 widzieliśmy na przykład, jak można oddzielić konstrukcję programu korzystającego z liczb wymiernych od implementacji liczb

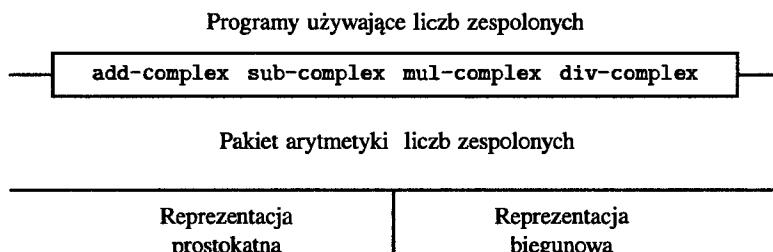
wymiernych, stosując dostępne w języku programowania pierwotne mechanizmy służące do tworzenia danych złożonych. Kluczowym pomysłem było tu wzniesienie bariery abstrakcji — w tym przypadku złożonej z selektorów i konstruktorów liczb wymiernych (`make-rat`, `numer` i `denom`) — oddzielającej sposób użycia liczb wymiernych od kryjącej się za nimi reprezentacji listowej. Podobna bariera abstrakcji oddziela szczegóły implementacji procedur wykonujących operacje arytmetyczne na liczbach wymiernych (`add-rat`, `sub-rat`, `mul-rat` i `div-rat`) od procedur „wyższego poziomu” operujących na liczbach wymiernych. Powstały w ten sposób program ma taką strukturę jak pokazana na rys. 2.1.

Takie bariery abstrakcji danych są potężnym narzędziem pozwalającym na kontrolę stopnia skomplikowania programów. Wydzielając reprezentacje kryjące się za obiektami danych, możemy podzielić zadanie skonstruowania dużego programu na mniejsze zadania, które mogą być wykonane niezależnie. Jednak taki rodzaj abstrakcji danych nie jest dostatecznie silny, gdyż nie zawsze jest sens mówić o „reprezentacji kryjącej się za” obiektem danych.

Przede wszystkim może istnieć wiele użytecznych reprezentacji obiektu danych i możemy chcieć skonstruować system, który potrafi operować na różnorodnych reprezentacjach. Dla prostego przykładu, liczby zespolone mogą być reprezentowane na dwa prawie równoważne sposoby: w układzie prostokątnym (część rzeczywista i urojona) lub biegunowym (wartość bezwzględna i kąt). Czasami bardziej odpowiednia jest postać prostokątna, a czasami biegunowa. Istotnie, możemy sobie łatwo wyobrazić system, w którym liczby zespolone są reprezentowane na obydwa sposoby i w którym procedury operujące liczbami zespolonymi działają na obydwu reprezentacjach.

Co ważniejsze, oprogramowanie jest często tworzone przez wielu ludzi pracujących przez dłuższy czas, przy wymaganiach zmieniających się w miarę upływu czasu. W takich warunkach jest po prostu niemożliwe, aby wszyscy zgodzili się z góry co do wyboru reprezentacji danych. Tak więc oprócz barier abstrakcji danych, które oddzielają reprezentację od jej użycia, potrzebujemy barier abstrakcji oddzielających od siebie różne decyzje projektowe i umożliwiających ich współistnienie w jednym programie. Ponadto ze względu na to, że duże programy często powstają przez połączenie wcześniej istniejących modułów zaprojektowanych niezależnie od siebie, potrzebujemy konwencji pozwalającej programistom na włączanie modułów do dużych systemów w sposób *addytywny*, tzn. nie wymagający zmian w ich projektach lub implementacji.

W niniejszym podrozdziale dowiemy się, jak sobie radzić z danymi, które w różnych częściach programu mogą być reprezentowane na różne sposoby. Wymaga to stworzenia *procedur ogólnych* (ang. *generic procedures*) — procedur mogących operować na danych, które mogą być reprezentowane na więcej niż jeden sposób. Nasza główna technika budowania procedur ogólnych będzie polegać na operowaniu obiektami danych ze *znacznikami typu* (ang. *type tags*),



Struktury listowe i pierwotna arytmetyka maszyny

Rys. 2.19. Bariery abstrakcji danych w systemie liczb zespolonych

tzn. obiektami danych, które zawierają informację określającą wprost, jak należy je przetwarzać. Omówimy również technikę programowania *sterowanego danymi* (ang. *data-directed programming*) — potężną i wygodną strategię implementacyjną służącą do addytywnego składania systemów z opercjami ogólnymi.

Zaczniemy od prostego przykładu liczb zespolonych. Zobaczmy, jak znaczniki typu i styl programowania sterowanego danymi umożliwiają konstruowanie osobno prostokątnej i biegunowej reprezentacji liczb zespolonych przy jednoczesnym zachowaniu pojęcia abstrakcyjnego obiektu „liczby zespolonej”. Dokonamy tego, definiując procedury arytmetyczne dla liczb zespolonych (add-complex, sub-complex, mul-complex i div-complex) przy zastosowaniu ogólnych selektorów dających dostęp do części liczb zespolonych w sposób niezależny od ich reprezentacji. Powstały w ten sposób system liczb zespolonych, jak to widać na rys. 2.19, zawiera dwa różne rodzaje barier abstrakcji. „Poziome” bariery abstrakcji odgrywają taką samą rolę jak te pokazane na rys. 2.1. Oddzielają one operacje „wyższego poziomu” od reprezentacji „niższego poziomu”. Ponadto na rysunku widać „pionową” barierę umożliwiającą niezależne konstruowanie i instalowanie alternatywnych reprezentacji.

W podrozdziale 2.5 pokażemy, jak zastosować znaczniki typów i programowanie sterowane danymi do opracowania uniwersalnego pakietu arytmetycznego. Będzie on zawierał procedury (add, mul itd.), które mogą operować na wszystkich rodzajach „liczb” i które można łatwo rozszerzyć, gdy będzie potrzebny nowy rodzaj liczb. W punkcie 2.5.3 powiemy zaś, jak zastosować arytmetykę ogólną w systemie wykonującym symboliczne operacje algebraiczne.

#### 2.4.1. Reprezentacje liczb zespolonych

Operujemy system wykonujący operacje arytmetyczne na liczbach zespolonych w ramach prostego, aczkolwiek nirealistycznego przykładu programu, który korzysta z operacji ogólnych. Na początek omówimy dwie dogodne repre-

zentacje liczb zespolonych w postaci par uporządkowanych: postać prostokątną (złożoną z części rzeczywistej i urojonej) oraz biegunową (złożoną z wartości bezwzględnej i kąta)<sup>43</sup>. W punkcie 2.4.2 pokażemy, jak obie te reprezentacje mogą współistnieć w jednym systemie dzięki zastosowaniu znaczników typu i operacji ogólnych.

Liczby zespolone, tak jak liczby wymierne, w naturalny sposób reprezentuje się jako pary uporządkowane. Zbiór liczb zespolonych możemy sobie wyobrazić jako dwuwymiarową przestrzeń z dwiema prostopadłymi osiami — osią „rzeczywistą” i „urojoną” (zob. rys. 2.20). Z takiego punktu widzenia liczbę zespoloną  $z = x + iy$  (gdzie  $i^2 = -1$ ) możemy sobie wyobrazić jako punkt na płaszczyźnie, którego współrzędna rzeczywista jest równa  $x$ , a współrzędna urojona jest równa  $y$ . W tej reprezentacji dodawanie liczb zespolonych sprawdza się do dodawania współrzędnych:

$$\text{część-rzeczywista}(z_1 + z_2) = \text{część-rzeczywista}(z_1) + \text{część-rzeczywista}(z_2)$$

$$\text{część-urojona}(z_1 + z_2) = \text{część-urojona}(z_1) + \text{część-urojona}(z_2)$$

Przy mnożeniu liczb zespolonych bardziej naturalne jest przedstawienie ich w reprezentacji biegunowej — jako wartość bezwzględna i kąt ( $r$  i  $\alpha$  z rys. 2.20). Iloczyn dwóch liczb zespolonych jest wektorem otrzymanym przez rozcięcie jednej liczby zespolonej proporcjonalnie do długości drugiej liczby, a następnie obrócenie jej o kąt drugiej liczby:

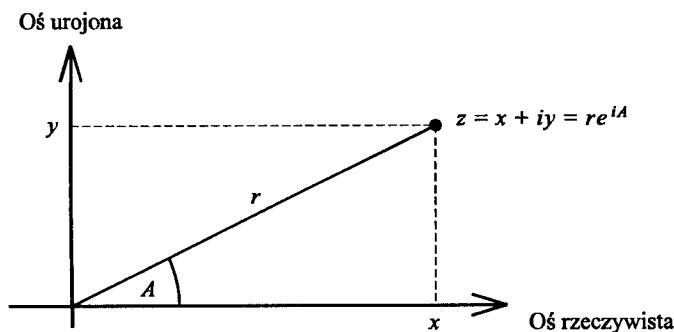
$$\text{wartość-bezwzględna}(z_1 \cdot z_2) =$$

$$\text{wartość-bezwzględna}(z_1) \cdot \text{wartość-bezwzględna}(z_2)$$

$$\text{kąt}(z_1 \cdot z_2) = \text{kąt}(z_1) + \text{kąt}(z_2)$$

Tak więc istnieją dwie różne reprezentacje liczb zespolonych — każda z nich jest odpowiednia dla innych operacji. Mimo to z punktu widzenia osoby piszącej program, który korzysta z liczb zespolonych, zasada abstrakcji danych sugeruje, że wszystkie operacje na liczbach zespolonych powinny być tak samo dostępne bez względu na reprezentację użytą przez komputer. Często na przykład przydaje się możliwość obliczenia wartości bezwzględnej liczby zespolonej określonej za pomocą współrzędnych prostokątnych. Podobnie, często przydatne jest określanie części rzeczywistej liczby zespolonej podanej za pomocą współrzędnych biegunowych.

<sup>43</sup> W prawdziwych systemach obliczeniowych preferowana jest zwykle postać prostokątną, a nie biegunową, ze względu na błędy zaokrągleń przy konwersji między postacią prostokątną i biegunową. Dlatego też przykład liczb zespolonych jest nierealistyczny. Niemniej jednak stanowi on przejrzystą ilustrację konstruowania systemu przy użyciu operacji ogólnych i dobry wstęp do bardziej pokaznych systemów, które będziemy tworzyć w dalszej części niniejszego rozdziału.



Rys. 2.20. Liczby zespolone jako punkty na płaszczyźnie

Konstruując taki system, możemy postępować zgodnie z tą samą strategią abstrakcji danych, którą stosowaliśmy, budując pakiet liczb wymiernych w punkcie 2.1.1. Założymy, że operacje na liczbach zespolonych są zaimplementowane za pomocą czterech selektorów: `real-part` (część rzeczywista), `imag-part` (część urojona), `magnitude` (wartość bezwzględna) i `angle` (kąt). Założymy również, że mamy dwie procedury konstruujące liczby zespolone: `make-from-real-imag`, która tworzy liczbę zespoloną o określonej części rzeczywistej i urojonej, oraz `make-from-mag-ang`, która tworzy liczbę zespoloną o określonej wartości bezwzględnej i kącie. Procedury te mają taką własność, że dla dowolnej liczby zespolonej z zarówno

```
(make-from-real-imag (real-part z) (imag-part z))
```

jak i

(make-from-mag-ang (magnitude z) (angle z))

tworzą liczby zespolone równe z.

Za pomocą tych konstruktorów i selektorów możemy zaimplementować arytmetykę liczb zespolonych korzystającą z określonych przez nie „danych abstrakcyjnych”, dokładnie tak jak to robiliśmy dla liczb wymiernych w punkcie 2.1.1. Jak widać z poniższych definicji, liczby zespolone możemy dodawać i odejmować za pomocą ich części rzeczywistych i urojonych, a mnożyć i dzielić za pomocą ich wartości bezwzględnych i katów:

```
(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
    (+ (angle z1) (angle z2)))))

(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
    (- (angle z1) (angle z2))))
```

Chcąc uzupełnić pakiet liczb zespolonych, musimy wybrać reprezentację oraz zaimplementować konstruktory i selektory za pomocą pierwotnej arytmetyki i pierwotnych struktur listowych. Istnieją dwa oczywiste sposoby dokonania tego: możemy reprezentować liczby zespolone w „postaci prostokątnej” jako pary (część rzeczywista, część urojona) lub w „postaci biegunowej” jako pary (wartość bezwzględna, kąt). Która reprezentację powinniśmy wybrać?

Aby uczynić te dwa warianty bardziej realnymi, wyobraźmy sobie, że dwoje programistów, Ben Bajerbit i Liz P. Haker, niezależnie konstruują reprezentacje systemu liczb zespolonych. Ben wybiera prostokątną reprezentację liczb zespolonych. Przy takim wyborze określanie części rzeczywistej i urojonej liczby zespolonej jest proste, tak samo jak tworzenie liczby zespolonej na podstawie danej części rzeczywistej i urojonej. Do wyznaczenia wartości bezwzględnej i kąta lub do utworzenia liczby zespolonej na podstawie danej wartości bezwzględnej i kąta używa on wzorów trygonometrycznych

$$\begin{aligned} x &= r \cos A & r &= \sqrt{x^2 + y^2} \\ y &= r \sin A & A &= \text{arc tg}(y, x) \end{aligned}$$

wiązących część rzeczywistą i urojoną  $(x, y)$  z wartością bezwzględną i kątem  $(r, A)$ <sup>44</sup>. Zatem reprezentacja Bena jest określona za pomocą następujących selektorów i konstruktorów:

```
(define (real-part z) (car z))

(define (imag-part z) (cdr z))

(define (magnitude z)
  (sqrt (+ (square (real-part z)) (square (imag-part z)))))

(define (angle z)
  (atan (imag-part z) (real-part z)))

(define (make-from-real-imag x y) (cons x y))

(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
```

<sup>44</sup> Użyta tutaj funkcja arcus tangens, obliczana przez procedurę `atan` języka Scheme, jest tak zdefiniowana, że jej wartością dla dwóch argumentów  $y$  i  $x$  jest kąt, którego tangens wynosi  $y/x$ . Znaki argumentów wyznaczają ćwiartkę układu współrzędnych, w której leży kąt.

Liz, w przeciwieństwie do Bena, wybrała biegunową reprezentację liczb zespolonych. Dla niej określenie wartości bezwzględnej i kąta jest proste, ale żeby uzyskać część rzeczywistą i urojoną, musi użyć funkcji trygonometrycznych. Oto reprezentacja Liz:

```
(define (real-part z)
  (* (magnitude z) (cos (angle z)))))

(define (imag-part z)
  (* (magnitude z) (sin (angle z))))

(define (magnitude z) (car z))

(define (angle z) (cdr z))

(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))

(define (make-from-mag-ang r a) (cons r a))
```

Zasada abstrakcji danych gwarantuje, że ta sama implementacja procedur `add-complex`, `sub-complex`, `mul-complex` i `div-complex` będzie działać zarówno dla reprezentacji Bena, jak i dla reprezentacji Liz.

## 2.4.2. Dane ze znacznikami

Jednym ze sposobów, w jaki można patrzeć na zasadę abstrakcji danych, jest zastosowanie zasady „minimalnego zobowiązania”. Implementując system liczb zespolonych (omówiony w punkcie 2.4.1), możemy skorzystać zarówno z reprezentacji prostokątnej Bena, jak i z reprezentacji biegunowej Liz. Bariera abstrakcji, jaką stanowią selektory i konstruktory, pozwala nam odwlec do ostatniej chwili wybór konkretnej reprezentacji naszych obiektów danych, a przez to umożliwia zachowanie maksymalnej elastyczności konstrukcji naszego systemu.

Stosowanie zasady minimalnego zobowiązania można posunąć jeszcze dalej. Jeśli zechcemy, możemy zachować niejednoznaczność reprezentacji nawet po utworzeniu selektorów i konstruktorów i zdecydować się na zastosowanie obu reprezentacji — Bena i Liz. Jeśli jednak w systemie występują obydwie reprezentacje, to potrzebna jest jakaś metoda rozróżnienia danych w postaci biegunowej i prostokątnej. W przeciwnym razie nie wiedzielibyśmy na przykład, jaki powinien być wynik procedury `magnitude` dla pary  $(3, 4)$  — czy powinien on być równy 5 (w przypadku reprezentacji biegunowej), czy też 3 (w przypadku reprezentacji prostokątnej). Prostym sposobem rozróżnienia reprezentacji jest dodanie do każdej liczby zespolonej *znacznika typu* — symbolu `rectangular` (dla reprezentacji prostokątnej) lub `polar` (dla reprezen-

tacji biegunowej). Dzięki temu, kiedy przychodzi nam działać na liczbach zespolonych, możemy za pomocą takiego znacznika stwierdzić, jak dana liczba zespolona jest reprezentowana, i zastosować odpowiednie selektory.

Chcąc operować na danych ze znacznikami, zakładamy, że są dostępne procedury `type-tag` i `contents`, wydobywające z obiektów danych znacznik typu i faktyczną zawartość (w przypadku liczb zespolonych — współrzędne prostokątne lub biegunowe). Wprowadzamy również procedurę `attach-tag`, która na podstawie znacznika typu i zawartości tworzy obiekt danych ze znacznikiem. Można to prosto zaimplementować za pomocą zwykłej struktury listowej:

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Złe dane ze znacznikiem -- TYPE-TAG" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Złe dane ze znacznikiem -- CONTENTS" datum)))
```

Korzystając z tych procedur, możemy zdefiniować predykaty `rectangular?` i `polar?` rozpoznające, odpowiednio, współrzędne prostokątne i biegunowe:

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))

(define (polar? z)
  (eq? (type-tag z) 'polar))
```

Stosując znaczniki typu, Ben i Liz mogą tak zmodyfikować swój kod, że ich dwie różne reprezentacje mogą współistnieć w jednym systemie. Ilekroć Ben tworzy liczbę zespoloną, zaznacza, iż jest ona reprezentowana w prostokątnym układzie współrzędnych. Ilekroć zaś Liz tworzy liczbę zespoloną, oznacza ją jako reprezentowaną w układzie biegunowym. Ponadto Ben i Liz muszą się upewnić, że nazwy ich procedur nie kolidują ze sobą. Mogą to osiągnąć, dodając odpowiednie końcówki do nazw procedur swoich reprezentacji — `rectangular` w przypadku Ben'a i `polar` w przypadku Liz. Oto poprawiona prostokątna reprezentacja Ben'a z punktu 2.4.1:

```
(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))
```

```

(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
            (square (imag-part-rectangular z)))))

(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))

(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))

(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))


```

a oto poprawiona biegunowa reprezentacja Liz:

```

(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z)))))

(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z)))))

(define (magnitude-polar z) (car z))

(define (angle-polar z) (cdr z))

(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x)))))

(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))


```

Każdy selektor ogólny jest zaimplementowany jako procedura sprawdzająca znacznik typu swojego argumentu i wywołująca odpowiednią procedurę operującą na danych tego typu. Aby na przykład uzyskać część rzeczywistą liczby zespolonej, procedura `real-part` bada znacznik typu i zależnie od jego wartości wywołuje procedurę Beno `real-part-rectangular` lub procedurę Liz `real-part-polar`. W obu przypadkach używamy procedury `contents` do wydobycia gołych danych, czyli pozbawionych znacznika, i przekazania ich, w zależności od potrzeby, do procedury działającej w układzie prostokątnym lub biegunowym:

```

(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)

```

```

(real-part-polar (contents z)))
(else (error "Nieznany typ -- REAL-PART" z)))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Nieznany typ -- IMAG-PART" z)))))

(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Nieznany typ -- MAGNITUDE" z)))))

(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Nieznany typ -- ANGLE" z)))))


```

Przy implementacji operacji arytmetycznych na liczbach zespolonych możemy bez zmian skorzystać z procedur `add-complex`, `sub-complex`, `mul-complex` i `div-complex`, przedstawionych w punkcie 2.4.1, gdyż wywoływane przez nie selektory są ogólne i działają na obydwu reprezentacjach. Na przykład procedura `add-complex` będzie nadal miała następującą postać:

```

(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2)))))


```

Na koniec musimy wybrać, czy chcemy tworzyć liczby zespolone w reprezentacji Bena, czy w reprezentacji Liz. Rozsądna decyzja może być tworzenie liczb zespolonych w układzie prostokątnym, gdy mamy dane ich części rzeczywiste i urojone, a w układzie biegunowym, gdy mamy dane ich wartości bezwzględne i kąty:

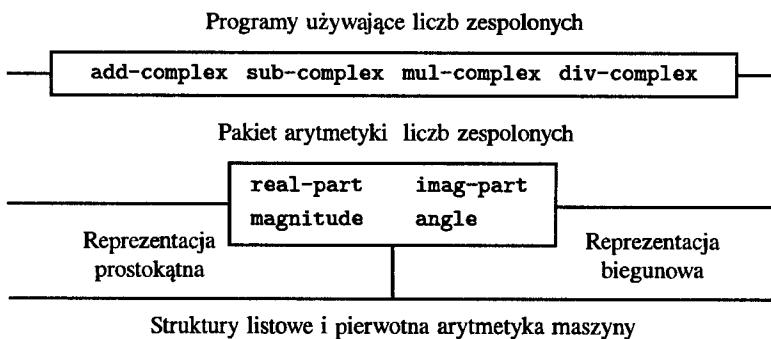
```

(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))

(define (make-from-mag-ang r a)
  (make-from-mag-ang-polar r a))


```

Powstały w ten sposób system liczb zespolonych ma strukturę taką, jak widać na rys. 2.21. System rozbito na trzy, stosunkowo niezależne części:



Rys. 2.21. Struktura ogólnego systemu arytmetyki liczb zespolonych

operacje arytmetyczne na liczbach zespolonych, biegunową implementację Liz oraz prostokątną implementację Bena. Ben i Liz mogliby niezależnie opracować implementacje biegunową i prostokątną, i obie mogłyby się kryć za abstrakcyjnym interfejsem konstruktorów i selektorów, używanym przez trzeciego programistę implementującego procedury arytmetyczne dla liczb zespolonych.

Ponieważ wszystkie obiekty danych mają znaczniki typów, selektory mogą na nich operować w sposób ogólny. Oznacza to, że każdy selektor jest tak zdefiniowany, że jego działanie zależy od konkretnego typu danych, na których operuje. Zwróćmy uwagę na ogólny mechanizm sprzągania różnych reprezentacji. W obrębie implementacji danej reprezentacji (np. pakietu biegunowego Liz) liczba zespolona jest parą bez znacznika typu (wartość bezwzględna, kąt). Gdy selektor ogólny operuje na danych typu polar, usuwa znacznik typu i przekazuje zawartość do kodu Liz. I odwrotnie, gdy Liz tworzy liczbę do ogólnego użytku, zaznacza ona jej typ, tak aby mógł on być odpowiednio rozpoznany przez procedury wyższego poziomu. Jak zobaczymy w podrozdziale 2.5, takie zasady postępowania, polegające na usuwaniu i dodawaniu znaczników przy przekazywaniu obiektów między poziomami, mogą stanowić ważną strategię organizacyjną.

### 2.4.3. Programowanie sterowane danymi i addytywność

Ogólna strategia polegająca na sprawdzaniu typów danych i wywoływaniu odpowiednich procedur jest nazywana *przekierowaniem ze względu na typ* (ang. *dispatching on type*). Jest to potężna strategia służąca do modularnego konstruowania systemów. Jednakże taka implementacja przekierowywania jak w punkcie 2.4.2 ma dwie istotne wady. Jedna z nich polega na tym, że ogólne procedury interfejsu (real-part, imag-part, magnitude i angle) muszą znać wszystkie możliwe reprezentacje. Założymy na przykład, że chcielibyśmy do naszego systemu liczb zespolonych wprowadzić nową reprezentację liczb zespolonych. Musielibyśmy w tym celu określić dla tej nowej reprezentacji typ

i do każdej ogólnej procedury interfejsu dodać klauzulę wychwytyującą ten typ i stosującą selektor odpowiedni dla tej reprezentacji.

Druga wada takiej techniki polega na tym, że chociaż poszczególne reprezentacje mogą być konstruowane niezależnie, to musimy zagwarantować, że w całym systemie nie pojawią się dwie różne procedury o tej samej nazwie. Dlatego właśnie Ben i Liz musieli zmienić oryginalne nazwy swoich procedur z punktu 2.4.1.

Przyczyną obu tych wad jest fakt, iż technika implementowania interfejsów ogólnych nie jest *addytywna*. Osoba implementująca procedury selektorów ogólnych musi je modyfikować za każdym razem, gdy instalowana jest nowa reprezentacja, a ludzie mający styczność z poszczególnymi reprezentacjami muszą modyfikować swój kod, aby uniknąć konfliktów nazw. W każdym z tych przypadków zmiany, jakich należy dokonać, są proste, ale muszą być wprowadzone, a to jest źródłem niedogodności i błędów. Nie stanowi to problemu w przypadku systemu liczb zespolonych jako takiego. Przypuśćmy jednak, że zamiast dwóch mamy setki różnych reprezentacji liczb zespolonych oraz że interfejs danych abstrakcyjnych obejmuje wiele ogólnych selektorów. Przyjmijmy, że żaden z programistów nie zna wszystkich procedur interfejsów ani wszystkich reprezentacji. Jest to rzeczywisty problem występujący w takich systemach jak systemy zarządzania bazami danych dużej skali.

Tym, czego nam potrzeba, jest jeszcze większa modularyzacja systemu. Można to osiągnąć, stosując technikę programistyczną znaną jako *programowanie sterowane danymi*. Aby zrozumieć, jak działa programowanie sterowane danymi, należy najpierw zauważać, że gdy mamy do czynienia ze zbiorem operacji ogólnych, które są wspólne dla różnych typów, wówczas mamy w rzeczywistości do czynienia z dwuwymiarową tablicą, w której jedna współrzędna określa możliwe operacje, a druga określa możliwe typy. W polach tablicy znajdują się procedury implementujące poszczególne operacje dla poszczególnych typów argumentów. W systemie liczb zespolonych, opracowanym w poprzednim punkcie, zależność między nazwą operacji, typem danych i faktyczną procedurą rozdzielono między klauzule warunkowe w ogólnych procedurach interfejsu. Jednak ta sama informacja może być przedstawiona w postaci tablicy, jak jest to pokazane na rys. 2.22.

Programowanie sterowane danymi jest techniką konstruowania programów działających bezpośrednio na takiej tablicy. Poprzednio zaimplementowaliśmy mechanizm sprzągający kod arytmetyki liczb zespolonych z dwiema reprezentacjami jako zestaw procedur, z których każda dokonywała jawnego przekierowania ze względu na typ. Teraz zaimplementujemy ten mechanizm (interfejs) w postaci pojedynczej procedury, która na podstawie nazwy operacji i typu argumentu wyszukuje w tablicy właściwą procedurę do zastosowania, a następnie stosuje ją do zawartości argumentu. Gdy to zrobimy, będziemy mogli

Operacje	Typy	
	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

Rys. 2.22. Tablica operacji dla systemu liczb zespolonych

dodawać do systemu nowe reprezentacje bez konieczności zmiany jakiejkolwiek istniejącej procedury — wystarczy tylko dodać nowe pola w tablicy.

Aby zaimplementować ten schemat, założymy, że mamy dwie procedury, put (wstaw) i get (pobierz), operujące na tablicy operacji i typów:

- (put *<op>* *<typ>* *<element>*)

wstawia do tablicy, w miejscu określonym przez operację *<op>* i typ *<typ>*, pole zawierające *<element>*;

- (get *<op>* *<typ>*)

wynikiem tej procedury jest zawartość pola tablicy określonego przez operację *<op>* i typ *<typ>*; jeśli takiego pola nie ma, to wynikiem jest wartość fałsz.

Chwilowo możemy założyć, że procedury put i get są elementami naszego języka. W rozdziale 3 (punkt 3.3.3) zobaczymy, jak można zaimplementować te i inne operacje na tablicach.

Oto jak programowanie sterowane danymi może być zastosowane w systemie liczb zespolonych. Ben, który opracował reprezentację prostokątną, implementuje swój kod tak, jak to zrobił pierwotnie. Definiuje on zestaw procedur, lub inaczej mówiąc *pakiet*, i sprząga go z resztą systemu, dodając w tablicy pola, które określają, jak system ma operować na liczbach w reprezentacji prostokątnej. Pakiet taki można zainstalować, wywołując następującą procedurę:

```
(define (install-rectangular-package)
  ;; procedury wewnętrzne
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
              (square (imag-part z)))))

  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a)))))

  (put 'real-part '(rectangular) (lambda (z) (car z)))
  (put 'imag-part '(rectangular) (lambda (z) (cdr z)))
  (put 'magnitude '(rectangular) (lambda (z) (sqrt (+ (square (real-part z))
                                         (square (imag-part z)))))))
  (put 'angle '(rectangular) (lambda (z) (atan (imag-part z) (real-part z)))))

  (put 'make-from-real-imag '(rectangular) (lambda (x y) (cons x y)))
  (put 'make-from-mag-ang '(rectangular) (lambda (r a) (cons (* r (cos a)) (* r (sin a))))))

  (put 'real-part '(polar) (lambda (z) (real-part (magnitude z) (angle z))))
  (put 'imag-part '(polar) (lambda (z) (imag-part (magnitude z) (angle z))))
  (put 'magnitude '(polar) (lambda (z) (magnitude z)))
  (put 'angle '(polar) (lambda (z) (angle z))))
```

*;; interfejs dla reszty systemu*

```
(define (tag x) (attach-tag 'rectangular x))
(put 'real-part '(rectangular) real-part)
(put 'imag-part '(rectangular) imag-part)
(put 'magnitude '(rectangular) magnitude)
(put 'angle '(rectangular) angle)
(put 'make-from-real-imag 'rectangular
     (lambda (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'rectangular
     (lambda (r a) (tag (make-from-mag-ang r a))))
'done)
```

Zwróćmy uwagę, że występujące tu procedury wewnętrzne są takie same jak te z punktu 2.4.1, które Ben napisał, pracując w odosobnieniu. Nie wymagają one żadnych zmian, aby móc je sprząć z resztą systemu. Co więcej, ponieważ definicje tych procedur znajdują się wewnętrz procedury instalacyjnej, Ben nie musi się martwić o konflikty nazw z innymi procedurami spoza jego pakietu. Aby sprząć swoje procedury z resztą systemu, Ben wstawia do tablicy procedurę `real-part` pod nazwą operacji `real-part` i typem `(rectangular)`; i podobnie dla pozostałych selektorów<sup>45</sup>. Interfejs definiuje również konstruktory, których powinien używać system zewnętrzny<sup>46</sup>. Są one takie same jak konstruktory zdefiniowane wewnętrznie przez Bena, z tym że dołączają jeszcze do obiektu znacznik typu.

Pakiet Liz, dla biegunowej reprezentacji liczb zespolonych, jest analogiczny:

```
(define (install-polar-package)
  ;; procedury wewnętrzne
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z)
    (* (magnitude z) (cos (angle z))))
  (define (imag-part z)
    (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interfejs dla reszty systemu
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part))
```

<sup>45</sup> Stosujemy listę `(rectangular)`, a nie symbol `rectangular`, aby dopuścić operacje wielo-argumentowe, których argumenty nie są wszystkie tego samego typu.

<sup>46</sup> Typ, pod którym są wstawiane konstruktory, nie musi być listą, gdyż wynikiem konstruktora jest zawsze obiekt jednego określonego typu.

```
(put 'imag-part '(polar) imag-part)
(put 'magnitude '(polar) magnitude)
(put 'angle '(polar) angle)
(put 'make-from-real-imag 'polar
      (lambda (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'polar
      (lambda (r a) (tag (make-from-mag-ang r a))))
'done)
```

Chociaż Ben i Liz nadal używają swoich pierwotnych procedur, zdefiniowanych pod takimi samymi nazwami (np. `real-part`), są one teraz zdefiniowane wewnętrz róznych procedur (zob. punkt 1.1.8), a zatem ich nazwy nie kolidują ze sobą.

Selektory arytmetyki liczb zespolonych mają dostęp do tablicy operacji i typów poprzez ogólną procedurę „operacji”, nazwaną `apply-generic`, która stosuje operację ogólną do argumentów. Procedura `apply-generic`, dla określonej nazwy operacji i typów argumentów, wyszukuje odpowiednie pole w tablicy i stosuje znajdującą się w nim procedurę, jeśli takowa istnieje<sup>47</sup>:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "Brak metody dla typów argumentów -- APPLY-GENERIC"
            (list op type-tags))))))
```

Korzystając z `apply-generic`, możemy następująco zdefiniować nasze selektory ogólne:

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

Zwróćmy uwagę, że nie zmienią się one wcale, jeśli do systemu dodamy nową reprezentację.

<sup>47</sup> Procedura `apply-generic` korzysta z notacji kropki i ogona, opisanej w ćwiczeniu 2.20, gdyż różne operacje ogólne mogą mieć różną liczbę argumentów. W `apply-generic` wartością `op` jest jej pierwszy argument, a wartością `args` jest lista pozostałych jej argumentów.

Apply-generic korzysta również z pierwotnej procedury `apply`, która ma dwa argumenty — procedurę i listę. Stosuje ona daną procedurę do argumentów z danej listy. Na przykład wynikiem

```
(apply + (list 1 2 3 4))
jest 10.
```

Z tablicy możemy również wydobyć konstruktory, jakich mają używać programy spoza pakietu do tworzenia liczb zespolonych na podstawie ich części rzeczywistych i urojonych lub na podstawie ich wartości bezwzględnych i kątów. Tak jak w punkcie 2.4.2, ilekroć znamy część rzeczywistą i urojoną, tworzymy liczbę w reprezentacji prostokątnej, a ilekroć znamy wartość bezwzględną i kąt, tworzymy liczbę w reprezentacji biegunowej:

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))

(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

### Ćwiczenie 2.73

W punkcie 2.3.2 był opisany program wykonujący różniczkowanie symboliczne:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp)))))
        (tu może być dodanych więcej reguł)
        (else (error "Nieznany rodzaj wyrażenia -- DERIV" exp))))
```

Możemy spojrzeć na ten program jak na program dokonujący przekierowania ze względu na rodzaj różniczkowanego wyrażenia. W takiej sytuacji „znacznikiem typu” danych jest symbol operatora algebraicznego (taki jak +), a wykonywaną operacją jest deriv. Możemy przekształcić ten program zgodnie ze stylem programowania sterowanego danymi, zmieniając podstawową procedurę różniczkowania w następujący sposób:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp)
                var)))))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))
```

- (a) Wyjaśnij, co zrobiono powyżej. Dlaczego nie można objąć przekierowaniem sterowanym danymi procedur `number?` i `variable?`?

- (b) Napisz procedury liczące pochodne sum i iloczynów oraz pomocniczy kod potrzebny do umieszczenia ich w tablicy używanej w powyższym programie.
- (c) Wybierz dowolną ulubioną regułę różniczkowania, na przykład taką jak reguła dla potęgowania (ćwiczenie 2.56), i zainstaluj ją w tym systemie sterowanym danymi.
- (d) W tym prostym manipulatorze algebraicznym typem wyrażenia jest wiążący je operator algebraiczny. Przypuśćmy jednak, że chcemy indeksować procedury odwrotnie, tak aby przekierowanie w procedurze `deriv` wyglądało następująco:

```
((get (operator exp) 'deriv) (operands exp) var)
```

Jakich zmian należy w związku z tym dokonać w systemie przekierowującym?

### Ćwiczenie 2.74

Nienasycone Przedsiębiorstwo SA to zdecentralizowane przedsiębiorstwo złożone z wielu niezależnych oddziałów rozmieszczonych po całym świecie. Sprzęt komputerowy przedsiębiorstwa połączono właśnie za pomocą połączeń sieciowych w bardzo pomysłową sieć, która każdemu użytkownikowi jawi się, jakby była jednym komputerem. Pani prezes Nienasyconego Przedsiębiorstwa, próbując po raz pierwszy skorzystać z możliwości sieci w celu uzyskania informacji administracyjnych z akt oddziałów, odkryła ze zdziwieniem, że choć wszystkie akta oddziałów zaimplementowano w postaci struktur danych w języku Scheme, jednak użyte struktury danych są różne w różnych oddziałach. Pospiesznie zwołano spotkanie kierowników oddziałów, aby znaleźć strategię zintegrowania akt w sposób satysfakcyjujący centralę przy jednoczesnym zachowaniu istniejącej autonomii oddziałów.

Pokaż, jak taką strategię można zaimplementować za pomocą programowania sterowanego danymi. Dla przykładu załóżmy, że akta osobowe personelu każdego oddziału stanowią pojedynczą kartotekę, która zawiera szereg rekordów identyfikowanych według nazwisk pracowników. Struktura takiej kartoteki różni się między oddziałami. Co więcej, akta osobowe każdego pracownika same są rodzajem kartoteki (o różnej strukturze dla różnych oddziałów) zawierającej informacje rozróżniane za pomocą identyfikatorów, takich jak `address` (adres) i `salary` (zarobki). W szczególności:

- (a) Zaimplementuj dla centrali procedurę `get-record`, która wyszukuje akta osobowe określonego pracownika w danej kartotece. Procedura ta powinna mieć zastosowanie do kartotek wszystkich oddziałów. Wyjaśnij, jaką strukturę powinny mieć kartoteki poszczególnych oddziałów. W szczególności, jaka informacja o typach powinna być dostępna?
- (b) Zaimplementuj dla centrali procedurę `get-salary`, której wynikiem jest informacja o pensji pracownika pochodzącej z danych akt osobowych, które mogą znajdować się w kartotece dowolnego oddziału. Jaką strukturę powinny mieć akta osobowe, aby ta operacja działała?
- (c) Zaimplementuj dla centrali procedurę `find-employee-record`. Powinna ona przeszukać kartoteki wszystkich oddziałów w poszukiwaniu akt osobowych danego pracownika. Jej wynikiem powinny być znalezione akta. Załóż, że argumentami tej procedury są nazwisko pracownika i lista kartotek wszystkich oddziałów.
- (d) Jakie zmiany należy wprowadzić, aby włączyć nowe akta osobowe do centralnego systemu, gdy Nienasycone Przedsiębiorstwo SA przejmuje kolejne przedsiębiorstwo?

## Przekazywanie komunikatów

Kluczowy pomysł w programowaniu sterowanym danymi polega na realizowaniu operacji ogólnych w programach za pomocą bezpośredniego operowania na tablicach operacji i typów, takich jak tablica pokazana na rys. 2.22. Przy wykorzystaniu stylu programowania z punktu 2.4.2 proces przekierowania ze względu na typ był zorganizowany w ten sposób, że każda operacja ogólna dbała o własne przekierowania. W rezultacie tablicę operacji i typów podzielono na wiersze, a każda procedura ogólna reprezentowała jeden wiersz tablicy.

Alternatywna strategia implementacyjna polega na rozbiciu tablicy na kolumny i na używaniu, zamiast „inteligentnych operacji”, które przekierowują ze względu na typy danych, „inteligentnych obiektów danych”, które przekierowują operacje według ich nazwy. Możemy to zrobić w ten sposób, że obiekt danych, taki jak liczba zespolona w reprezentacji prostokątnej, jest reprezentowany jako procedura, której argumentem jest odpowiednia nazwa operacji i która wykonuje wskazaną operację. Przy takim podejściu procedurę `make-from-real-imag` można by zapisać następująco:

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else
           (error "Nieznana operacja -- MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

Odpowiednia procedura `apply-generic`, która stosuje operację ogólną do argumentu, po prostu przekazuje nazwę operacji do obiektu danych i pozwala obiekowi ją wykonać<sup>48</sup>:

```
(define (apply-generic op arg) (arg op))
```

Zwrócmy uwagę, że wynikiem `make-from-real-imag` jest procedura — wewnętrzna procedura `dispatch`. Jest ona wywoływana, gdy `apply-generic` zleca wykonanie operacji.

Taki styl programowania jest nazywany *przekazywaniem komunikatów*. Nazwa bierze się stąd, iż można sobie wyobrazić, że obiekt danych jest jednostką otrzymującą nazwy zleczanych operacji w postaci „komunikatów”. Przykład takiego przekazywania komunikatów widzieliśmy już w punkcie 2.1.3, gdzie

<sup>48</sup> Takie podejście charakteryzuje się pewnym ograniczeniem — pozwala na zrealizowanie tylko jednoargumentowych operacji ogólnych.

cons, car i cdr mogły być zdefiniowane bez użycia obiektów danych, wyłącznie za pomocą procedur. Widzimy teraz, że przekazywanie komunikatów nie jest tylko matematyczną sztuczką, ale użyteczną techniką organizowania systemów zawierających operacje ogólne. W pozostałej części niniejszego rozdziału, omawiając ogólne operacje arytmetyczne, wróćmy do programowania sterowanego danymi, odkładając na bok przekazywanie komunikatów. W rozdziale 3 ponownie zajmiemy się przekazywaniem komunikatów i zobaczymy, że może być ono potężnym narzędziem konstruowania programów symulacyjnych.

### Ćwiczenie 2.75

Zaimplementuj konstruktor `make-from-mag-ang`, stosując styl programowania oparty na przekazywaniu komunikatów. Powstała procedura powinna być analogiczna do podanej wcześniej `make-from-real-imag`.

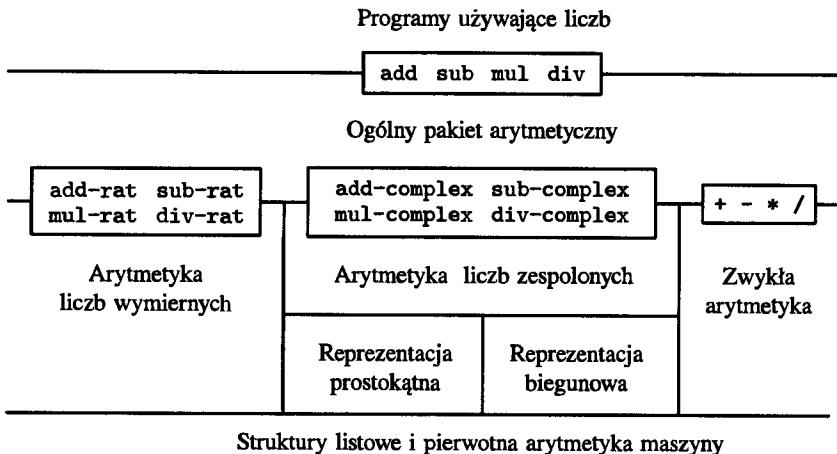
### Ćwiczenie 2.76

Duże systemy zawierające operacje ogólne rozwijają się, potrzebne są zatem nowe typy obiektów danych lub nowe operacje. Dla każdej z trzech strategii — operacji ogólnych z jawnym przekierowaniem, programowania sterowanego danymi i przekazywania komunikatów — opisz zmiany, których trzeba dokonać w systemie, aby dodać nowe typy lub nowe operacje. Która ze strategii byłaby najodpowiedniejsza dla systemu, do którego często dodaje się nowe typy, a która dla systemu, do którego często dodaje się nowe operacje?

## 2.5. Systemy z operacjami ogólnymi

W poprzednim podrozdziale zobaczyliśmy, jak konstruować systemy, w których obiekty danych mogą być reprezentowane na więcej niż jeden sposób. Kluczowy pomysł polegał na łączeniu kodu określającego operacje na danych z kilkoma reprezentacjami za pomocą ogólnych procedur interfejsu. Teraz dowiemy się, jak zastosować ten sam pomysł nie tylko do definiowania operacji, które są ogólne ze względu na różne reprezentacje, lecz także do definiowania operacji, które są ogólne ze względu na różne rodzaje argumentów. Widzieliśmy już kilka różnych pakietów operacji arytmetycznych: arytmetykę pierwotną (`+, -, *, /`) wbudowaną w nasz język, arytmetykę liczb wymiernych (`add-rat`, `sub-rat`, `mul-rat`, `div-rat`) z punktu 2.1.1 oraz arytmetykę liczb zespolonych zaimplementowaną w punkcie 2.4.3. Zastosujemy teraz technikę programowania sterowanego danymi do zbudowania pakietu operacji arytmetycznych, który zawiera wszystkie dotychczas przez nas poznane pakiety arytmetyczne.

Na rysunku 2.23 widać strukturę systemu, który zbudujemy. Zwrócić uwagę na bariery abstrakcji. Z punktu widzenia kogoś używającego „liczb” jest jedna procedura `add` działająca na liczbach dowolnego rodzaju. `Add` jest częścią ogólnego interfejsu umożliwiającego programom korzystającym z liczb jednoznaczny dostęp do trzech odrębnych pakietów: zwykłej arytmetyki, arytmetyki liczb



Rys. 2.23. Ogólny system arytmetyczny

wymiernych i arytmetyki liczb zespolonych. Poszczególne pakiety arytmetyczne (np. pakiet liczb zespolonych) są dostępne poprzez procedury ogólne (takie jak `add-complex`), które łączą pakiety skonstruowane dla różnych reprezentacji (takich jak reprezentacja prostokątna i biegunowa). Co więcej, struktura systemu jest addytywna, czyli poszczególne pakiety arytmetyczne mogą być konstruowane niezależnie i łączone w jeden ogólny system arytmetyczny.

### 2.5.1. Ogólne operacje arytmetyczne

Konstruowanie ogólnych operacji arytmetycznych jest analogiczne do konstruowania ogólnych operacji na liczbach zespolonych. Chcielibyśmy na przykład dysponować ogólną procedurą dodawania `add`, która dla zwykłych liczb działa jak zwykłe dodawanie `+`, dla liczb wymiernych jak `add-rat`, a dla liczb zespolonych jak `add-complex`. Możemy zaimplementować `add` i inne ogólnie operacje arytmetyczne za pomocą tej samej strategii, którą zastosowaliśmy w punkcie 2.4.3 do zaimplementowania selektorów ogólnych dla liczb zespolonych. Do każdego rodzaju liczb będziemy dołączać znacznik typu, a procedury ogólne będą przekierowywać operacje do odpowiednich pakietów, stosownie do znaczników typów argumentów.

Ogólne procedury arytmetyczne definiujemy następująco:

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

Na początek zainstalujemy pakiet obsługujący *zwykłe* liczby, tzn. liczby pierwotne dostępne w naszym języku. Będziemy je oznaczać symbolem `scheme-`

-number. Operacje arytmetyczne należące do tego pakietu, to pierwotne procedury arytmetyczne (nie ma więc potrzeby definiowania dodatkowych procedur działających na liczbach bez znaczników). Ponieważ są to operacje dwuargumentowe, są one wstawiane do tablicy pod listą (scheme-number scheme-number):

```
(define (install-scheme-number-package)
  (define (tag x)
    (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number
       (lambda (x) (tag x)))
  'done)
```

Użytkownicy pakietu liczb pierwotnych języka Scheme będą tworzyć zwykłe liczby (ze znacznikami) za pomocą procedury

```
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```

Teraz, gdy schemat ogólnego systemu arytmetycznego jest gotowy, możemy łatwo dołączać nowe rodzaje liczb. Oto pakiet wykonujący operacje arytmetyczne na liczbach wymiernych. Zwróćmy uwagę, że dzięki addytywności możemy bez zmian użyć kodu z punktu 2.1.1 jako procedur wewnętrznych pakietu:

```
(define (install-rational-package)
  ;; procedury wewnętrzne
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  (define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
```

```

(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y)))))

;; interfejs dla reszty systemu
(define (tag x) (attach-tag 'rational x))
(put 'add '(rational rational)
     (lambda (x y) (tag (add-rat x y))))
(put 'sub '(rational rational)
     (lambda (x y) (tag (sub-rat x y))))
(put 'mul '(rational rational)
     (lambda (x y) (tag (mul-rat x y))))
(put 'div '(rational rational)
     (lambda (x y) (tag (div-rat x y))))
(put 'make 'rational
     (lambda (n d) (tag (make-rat n d))))
'done)

(define (make-rational n d)
  (get 'make 'rational) n d))

```

Możemy zainstalować podobny pakiet obsługujący liczby zespolone, stosując znacznik `complex`. Tworząc pakiet, wydobywamy z tablicy operacje `make-from-real-imag` i `make-from-mag-ang`, zdefiniowane w pakietach reprezentacji prostokątnej i biegunowej. Addytywność umożliwia nam użycie procedur `add-complex`, `sub-complex`, `mul-complex` i `div-complex` z punktu 2.4.1 jako operacji wewnętrznych.

```

(define (install-complex-package)
  ;; procedure importowane z pakietów reprezentacji
  ;; prostokątnej i biegunowej
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))

  ;; procedure wewnętrzne
  (define (add-complex z1 z2)
    (make-from-real-imag (+ (real-part z1) (real-part z2))
                         (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (- (real-part z1) (real-part z2))
                         (- (imag-part z1) (imag-part z2))))
)

```

```

(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
;; interfejs dla reszty systemu
(define (tag z) (attach-tag 'complex z))
(put 'add '(complex complex)
     (lambda (z1 z2) (tag (add-complex z1 z2))))
(put 'sub '(complex complex)
     (lambda (z1 z2) (tag (sub-complex z1 z2))))
(put 'mul '(complex complex)
     (lambda (z1 z2) (tag (mul-complex z1 z2))))
(put 'div '(complex complex)
     (lambda (z1 z2) (tag (div-complex z1 z2))))
(put 'make-from-real-imag 'complex
     (lambda (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'complex
     (lambda (r a) (tag (make-from-mag-ang r a))))
'done)

```

Programy spoza pakietu liczb zespolonych mogą tworzyć liczby zespolone, albo określając ich części rzeczywiste i urojone, albo określając ich wartości bezwzględne i kąty. Zwróćmy uwagę, jak kryjące się za tym procedury, zdefiniowane pierwotnie w pakietach reprezentacji prostokątnej i biegunowej, są eksportowane do pakietu liczb zespolonych, a stamtąd do otaczającego świata.

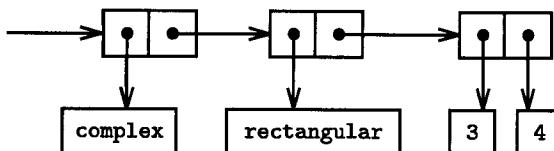
```

(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))

(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))

```

Mamy tu do czynienia z dwupoziomowym systemem znaczników typu. Tym powodem liczba zespolona, taka jak  $3 + 4i$ , jest reprezentowana w postaci prostokątnej tak, jak widać to na rys. 2.24. Zewnętrzny znacznik (`complex`) jest używany do skierowania liczby do pakietu liczb zespolonych. Wewnątrz tego pakietu kolejny znacznik (`rectangular`) jest używany do skierowania liczby do pakietu reprezentacji prostokątnej. W dużym i skomplikowanym systemie może być wiele takich poziomów, połączonych kolejno ze sobą za pomocą operacji ogólnych. Gdy obiekt danych jest przekazywany „w dół”, znacznik zewnętrzny jest używany do skierowania go do odpowiedniego pakietu i jest odrywany (za pomocą `contents`). Wówczas znacznik z kolejnego poziomu (jeśli istnieje) staje się widoczny i może być użyty do dalszego przekierowania.



Rys. 2.24. Reprezentacja prostokątna liczby  $3 + 4i$

W powyższych pakietach używaliśmy `add-rat`, `add-complex` i innych procedur arytmetycznych dokładnie tak, jak je pierwotnie napisano. Odkąd jednak są one zdefiniowane wewnątrz różnych procedur instalujących pakiety, nie muszą mieć różnych nazw — możemy je po prostu w obu pakietach nazwać `add`, `sub`, `mul` i `div`.

### Ćwiczenie 2.77

Ludwik Myślicielak próbuje obliczyć wartość wyrażenia (`magnitude z`), gdzie `z` jest obiektem pokazanym na rys. 2.24. Zaskoczony stwierdził, że zamiast wyniku równego 5 otrzymał komunikat o błędzie pochodzący z procedury `apply-generic`, mówiący, że brak metody realizującej operację `magnitude` na obiektach typu `(complex)`. Pokazał to Liz P. Haker, która powiedziała: „Problem polega na tym, że selektorów liczb zespolonych nigdy nie zdefiniowano dla liczb typu `complex`, a jedynie dla liczb ze znacznikami `polar` i `rectangular`. Aby całość zadziałała, musisz jedynie do pakietu `complex` dodać następujący fragment”:

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

Opisz szczegółowo, dlaczego to zadziała. Jako przykład prześledź wszystkie wywołania procedur mające miejsce przy obliczaniu wyrażenia (`magnitude z`), gdzie `z` jest obiektem pokazanym na rys. 2.24. W szczególności, ile razy jest wywoływana procedura `apply-generic`? Do jakiej procedury następuje przekierowanie w każdym z tych przypadków?

### Ćwiczenie 2.78

Procedury wewnętrzne w pakiecie `scheme-number` są w gruncie rzeczy niczym więcej jak wywołaniami procedur pierwotnych `+`, `-` itd. Nie można było użyć bezpośrednio operacji pierwotnych naszego języka, gdyż nasz system znaczników wymaga, żeby do każdego obiektu danych był przyczepiony znacznik typu. W rzeczywistości jednak wszystkie implementacje Lispu mają swój wewnętrzny system typów. Predykaty pierwotne, takie jak `symbol?` i `number?`, sprawdzają, czy obiekty danych są określonych typów. Zmień tak definicje procedur `type-tag`, `contents` i `attach-tag` (z punktu 2.4.2), aby nasz ogólny system korzystał z wewnętrznego systemu typów języka Scheme. Oznacza to, że system powinien działać tak jak poprzednio, przy czym zwykłe liczby powinny być reprezentowane po prostu jako liczby w języku Scheme, a nie jako pary, których `car` jest symbolem `scheme-number`.

**Ćwiczenie 2.79**

Zdefiniuj ogólny predykat równości `equ?`, który sprawdza, czy dwie liczby są równe, i zainstaluj go w ogólnym pakiecie arytmetycznym. Operacja ta powinna działać dla zwykłych liczb, liczb wymiernych i liczb zespolonych.

**Ćwiczenie 2.80**

Zdefiniuj ogólny predykat `=zero?`, który sprawdza, czy jego argument jest zerem, i zainstaluj go w ogólnym pakiecie arytmetycznym. Operacja ta powinna działać dla zwykłych liczb, liczb wymiernych i liczb zespolonych.

**2.5.2. Łączenie danych różnych typów**

Zobaczyliśmy, jak można zdefiniować ujednolicony system arytmetyczny, obejmujący zwykłe liczby, liczby zespolone, liczby wymierne i dowolny inny rodzaj liczb, które moglibyśmy wymyślić. Pomineliśmy jednak ważną kwestię. Zdefiniowane przez nas dotąd operacje traktują różne typy danych jako całkowicie niezależne. I tak do dodawania, powiedzmy, dwóch zwykłych liczb i dwóch liczb zespolonych służą oddzielne pakiety. Nie rozważaliśmy jeszcze tego, że definiowanie operacji wykraczających poza granice jednego typu, takich jak dodawanie liczb zespolonych do zwykłych liczb, może mieć sens. Wielkim wysiłkiem wprowadziliśmy bariery dzielące nasze programy na części, które mogą być opracowywane i rozumiane niezależnie. Chcielibyśmy ostrożnie i w kontrolowany sposób wprowadzić operacje działające równocześnie na obiektach różnych typów, nie naruszając przy tym w istotny sposób granic modułów.

Jeden ze sposobów wprowadzenia operacji działających równocześnie na obiektach różnych typów polega na skonstruowaniu osobnych procedur dla wszystkich możliwych kombinacji typów, dla których można wykonać daną operację. Moglibyśmy na przykład rozszerzyć pakiet liczb zespolonych tak, aby udostępniał procedurę dodającą liczby zespolone do zwykłych liczb, i umieścić ją w tablicy ze znacznikiem (`complex scheme-number`)<sup>49</sup>:

```
;; ten fragment należy dodać do pakietu liczb zespolonych
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x)
                       (imag-part z)))

(put 'add '(complex scheme-number)
     (lambda (z x) (tag (add-complex-to-schemenum z x))))
```

Technika ta działa, ale jest nieporęczna. Przy takim postępowaniu koszt wprowadzenia nowego typu obejmuje nie tylko stworzenie pakietu procedur

<sup>49</sup> Musimy również umieścić w tablicy prawie identyczną procedurę obsługującą typy (`scheme-number complex`).

dla tego typu, ale również stworzenie i zainstalowanie procedur implementujących operacje na obiektach różnych typów. Procedury takie z łatwością mogą obejmować dużo więcej kodu niż operacje na obiektach tylko danego typu. Metoda ta również podważa możliwość addytywnego łączenia niezależnych pakietów, a przynajmniej łamie ograniczenia zakresu, w jakim implementator jednego pakietu musi brać pod uwagę inne pakiety. Wydaje się sensowne, aby w przedstawionym powyżej przykładzie obsługa operacji mieszanych na liczbach zespolonych i zwykłych liczbach należała do obowiązków pakietu liczb zespolonych. Jednakże łączenie liczb wymiernych i zespolonych mogłoby być wykonywane przez pakiet liczb zespolonych, pakiet liczb wymiernych lub jakiś trzeci pakiet używający operacji udostępnianych przez te dwa pakiety. Formułowanie spójnych zasad podziału kompetencji między pakiety może być przytaczającym zadaniem w przypadku konstruowania systemów złożonych z wielu pakietów i zawierających wiele operacji na obiektach różnych typów.

### Rzutowanie typów

W ogólnej sytuacji, gdy mamy zupełnie ze sobą niezwiązane operacje działające na zupełnie ze sobą niezwiązanych typach, jawne implementowanie operacji na obiektach różnych typów, choć może być nieporęczne, jest najlepszym rozwiązaniem, na jakie możemy liczyć. Na szczęście zwykle możemy sobie lepiej radzić, korzystając z dodatkowych zależności ukrytych w strukturze typów naszego systemu. Często różne typy danych nie są całkowicie niezależne i mogą istnieć sposoby, dzięki którym obiekty jednego typu mogą być widziane, jakby były innego typu. Proces umożliwiający to nazywany jest *rzutowaniem typów*. Jeśli na przykład mamy wykonać działanie arytmetyczne na zwykłej liczbie i na liczbie zespolonej, to możemy przedstawić zwykłą liczbę jako liczbę zespoloną o części urojonej równej zeru. Tym samym sprowadzamy problem do wykonania operacji na dwóch liczbach zespolonych, co może być obsłużone w zwykły sposób przez pakiet arytmetyki liczb zespolonych.

Na ogół możemy ten pomysł wprowadzić w życie, konstruując procedury rzutowania typów, przekształcające obiekt jednego typu w równoważny mu obiekt innego typu. Oto typowa procedura rzutowania typów, która przekształca zwykłą liczbę w liczbę zespoloną o takiej samej części rzeczywistej i części urojonej równej zeru:

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

Takie procedury rzutowania typów umieszczamy w specjalnej tablicy rzutowania typów, której pola są indeksowane nazwami dwóch typów:

```
(put-coercion 'scheme-number 'complex scheme-number->complex)
```

(Zakładamy, że są dostępne procedury `put-coercion` i `get-coercion`, umożliwiające operowanie na tej tablicy). Zazwyczaj niektóre pola tablicy będą puste, gdyż na ogół nie można rzutować każdego typu na wszystkie inne typy. Nie da się na przykład rzutować dowolnej liczby zespolonej na zwykłą liczbę, więc w tablicy nie będzie żadnej procedury ogólniej `complex->scheme-number`.

Gdy już wstawimy procedury rzutowania do tablicy, możemy obsługiwać rzutowanie typów w jednolity sposób, modyfikując procedurę `apply-generic` z punktu 2.4.3. Kiedy mamy zastosować operację, sprawdzamy najpierw, tak jak poprzednio, czy operacja ta jest zdefiniowana dla typów argumentów. Jeśli tak, to przekierowujemy operację do procedury umieszczonej w tablicy operacji i typów. W przeciwnym razie próbujemy wykonać rzutowanie typów. Dla uproszczenia ograniczymy się do operacji dwuargumentowych<sup>50</sup>. Sprawdzamy w tablicy rzutowania typów, czy typ pierwszego argumentu może być rzutowany na typ drugiego argumentu. Jeśli tak, to rzutujemy pierwszy argument i ponownie próbujemy zastosować operację. Jeśli typ pierwszego argumentu w ogóle nie może być rzutowany na typ drugiego argumentu, to próbujemy wykonać rzutowanie w drugą stronę i sprawdzamy, czy można rzutować typ drugiego argumentu na typ pierwszego argumentu. W końcu, jeśli nie da się rzutować jednego z typów na drugi, rezygnujemy z wykonania operacji. Oto procedura:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                           (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                           (apply-generic op a1 (t2->t1 a2)))
                        (else
                           (error "Brak metody dla typów argumentów"
                                 (list op type-tags))))))
              (error "Brak metody dla typów argumentów"
                    (list op type-tags)))))))
```

<sup>50</sup> Ogólny przypadek jest przedmiotem ćwiczenia 2.82.

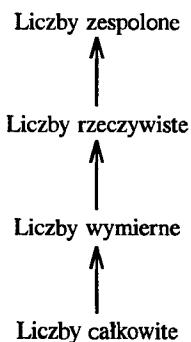
Taki schemat rzutowania typów ma wiele zalet w porównaniu z przedstawioną wcześniej metodą jawnego definiowania operacji na obiektach różnych typów. Chociaż nadal musimy pisać procedury rzutowania wiążące ze sobą różne typy (być może nawet  $n^2$  procedur w przypadku systemu zawierającego  $n$  typów), musimy jednak napisać tylko jedną procedurę dla każdej pary typów, a nie inną dla każdej pary typów i każdej ogólnej operacji<sup>51</sup>. Liczymy tutaj na to, że odpowiednie przekształcenia między typami zależą tylko od samych typów, a nie od operacji, jakie mamy wykonać.

Jednakże mogą istnieć zastosowania, dla których nasz schemat rzutowania typów nie jest dostatecznie ogólny. Nawet jeśli żaden z obiektów, na których mamy wykonać operację, nie może być przekształcony na typ drugiego obiektu, to nadal wykonanie operacji może być możliwe, jeżeli oba obiekty zostaną przekształcone w jakiś trzeci typ. Zwykle, aby radzić sobie z taką złożonością i zachować modularność programów, należy tworzyć systemy korzystające w jeszcze większym stopniu ze struktury zależności między typami, co omówimy dalej.

## Hierarchie typów

Przedstawiony powyżej schemat rzutowania typów opiera się na istnieniu naturalnej zależności między parami typów. Często istnieje bardziej „globalna” struktura zależności między poszczególnymi typami. Założymy na przykład, że tworzymy ogólny system arytmetyczny obsługujący liczby całkowite, wymierne, rzeczywiste i zespolone. W takim systemie całkiem naturalne wydaje się traktowanie liczb całkowitych jako szczególnego rodzaju liczb wymiernych, które z kolei są szczególnym rodzajem liczb rzeczywistych, które z kolei są szczególnym rodzajem liczb zespolonych. Mamy tu faktycznie do czynienia z tzw. *hierarchią typów*, w której na przykład liczby całkowite są *podtypem* (ang. *subtype*) liczb wymiernych (tzn. każda operacja, jaka może być zastosowana do liczby wymiernej, może automatycznie być zastosowana również do liczby całkowitej). I odwrotnie, mówimy, że liczby wymierne stanowią *nadtyp* (ang. *supertype*) liczb całkowitych. Ta konkretna hierarchia, z jaką mamy tu do czynienia, jest bardzo prosta. Każdy typ ma w niej co najwyżej jeden bezpośredni nadtyp i co najwyżej jeden bezpośredni podtyp. Taka struktura, nazywana *wieżą*, jest przedstawiona na rys. 2.25.

<sup>51</sup> Jeśli będziemy sprytni, to zwykle wystarczy nam mniej niż  $n^2$  rzutowań typów. Jeśli na przykład wiemy, jak zamienić wartości typu 1 w wartości typu 2 oraz jak zamienić wartości typu 2 w wartości typu 3, to możemy tego użyć do zamiany wartości typu 1 w wartości typu 3. Może to istotnie zmniejszyć liczbę rzutowań, jakie musimy określić, dodając nowy typ do systemu. Jeśli zechcemy wbudować w nasz system odpowiednio wyrafinowany mechanizm, to może on przeszukiwać „graf” zależności między typami i automatycznie tworzyć te procedury rzutowania, które można wywnioskować z jawnie podanych procedur.

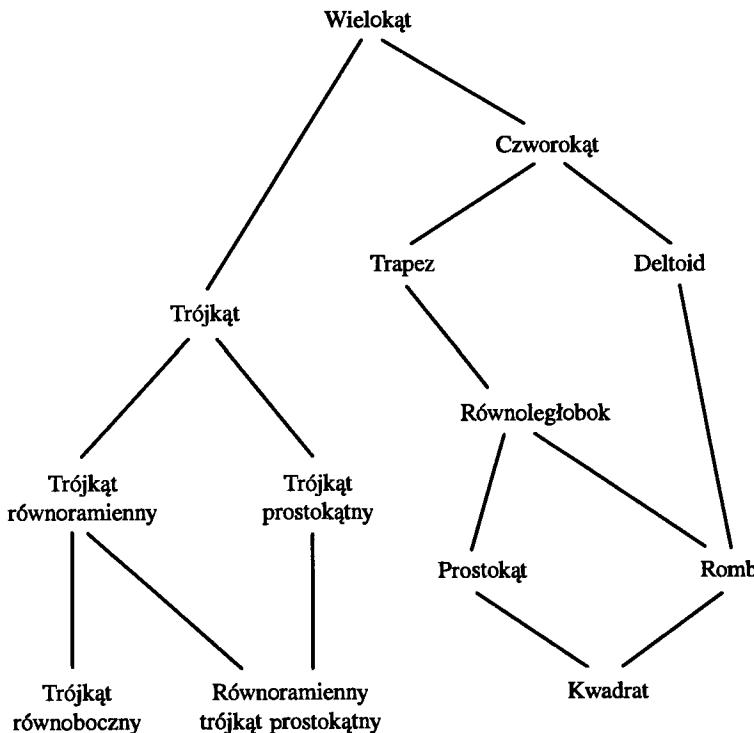


Rys. 2.25. Wieża typów

Jeśli mamy strukturę wieży, to możemy znacznie uprościć problem dodawania nowego typu do hierarchii, gdyż musimy jedynie określić, jak ten nowy typ jest osadzony w nadtypie znajdującym się bezpośrednio nad nim oraz w jaki sposób zawiera w sobie podtyp znajdujący się bezpośrednio pod nim. Jeśli na przykład chcemy dodać liczbę całkowitą do liczby zespolonej, to nie musimy jawnie definiować specjalnej procedury rzutowania `integer->complex`. Zamiast tego definiujemy, jak liczbę całkowitą zamienić na wymierną, jak liczbę wymierną zamienić na rzeczywistą i jak liczbę rzeczywistą zamienić na zespoloną. Pozwalamy wówczas systemowi na zamianę liczby całkowitej na zespoloną w trzech krokach, a następnie dodajemy do siebie dwie liczby zespolone.

Możemy zmodyfikować naszą procedurę `apply-generic` w następujący sposób. Dla każdego typu musimy zdefiniować procedurę `raise`, która rzutuje obiekty tego typu „w górę” o jeden poziom w wieży. Wówczas, gdy system ma wykonać operację na obiektach różnych typów, może kolejno rzutować obiekty niższych typów, aż wszystkie znajdą się na tym samym poziomie wieży. (Ćwiczenia 2.83 i 2.84 dotyczą szczegółów implementacji takiej strategii).

Inną zaletą wieży jest to, że możemy łatwo zaimplementować pojęcie „dziedziczenia” przez każdy typ operacji zdefiniowanych dla jego nadtypu. Na przykład, nawet jeśli nie określmy specjalnej procedury obliczającej część rzeczywistą liczby całkowitej, to mimo to możemy oczekwać, że procedura `real-part` jest określona na liczbach całkowitych z racji tego, że liczby całkowite są podtypem liczb zespolonych. W przypadku wieży typów możemy w jednolity sposób wprowadzić taki schemat rzutowania, modyfikując procedurę `apply-generic`. Jeśli potrzebna operacja nie jest bezpośrednio zdefiniowana dla typu danego obiektu, możemy rzutować obiekt na jego nadtyp i spróbować ponownie. W ten sposób wspinamy się w górę wieży, przekształcając przy tym argumenty, dopóki albo nie znajdziemy poziomu, na którym dana operacja może być wykonana, albo nie znajdziemy się na szczycie wieży (w tym przypadku rezygnujemy z wykonania operacji).



Rys. 2.26. Zależności między rodzajami figur geometrycznych

Jeszcze jedną zaletą wieży, w porównaniu z bardziej ogólnymi hierarchiami, jest to, że pozwala łatwo rzutować obiekty danych „w dół” do ich najprostszej reprezentacji. Jeśli na przykład dodajemy  $2 + 3i$  do  $4 - 3i$ , to dobrze byłoby uzyskać w wyniku liczbę całkowitą 6 zamiast liczby zespolonej  $6 + 0i$ . W ćwiczeniu 2.85 omawiamy sposób zaimplementowania takiej operacji rzutowania w dół. (Sztuczka polega na znalezieniu ogólnego sposobu odróżniania takich obiektów, które można rzutować w dół, jak na przykład  $6 + 0i$ , od takich, których nie można rzutować w dół, jak na przykład  $6 + 2i$ ).

### Niedostatki hierarchii typów

Jeśli typy danych w systemie można w naturalny sposób ułożyć w wieżę, to jak widzieliśmy, znacznie upraszcza to problem operacji ogólnych działających na argumentach różnych typów. Niestety, zwykle tak nie jest. Na rysunku 2.26 widać bardziej złożony układ różnych typów — tutaj przedstawiający zależności między różnymi rodzajami figur geometrycznych. Jak widać, zazwyczaj typ może mieć więcej niż jeden podtyp. Na przykład, zarówno trójkąty, jak i czworokąty są podtypami wielokątów. Ponadto typ może mieć więcej niż jeden nadtyp. Równoramienny trójkąt prostokątny może na przykład być traktowany albo jako trójkąt równoramienny, albo jako prostokątny. Kwestia wielu

nadtypów jest szczególnie kłopotliwa, gdyż oznacza, że istnieje wiele sposobów rzutowania typu „do góry” w hierarchii. Znalezienie „właściwego” nadtypu, w którym należy zastosować daną operację do obiektu, może wymagać od takiej procedury jak `apply-generic` znaczającej ilości poszukiwań w całym grafie typów. Ponieważ na ogół typ może mieć również wiele podtypów, podobny problem występuje przy rzutowaniu wartości „w dół” hierarchii typów. Radzenie sobie z wieloma wzajemnie zależnymi typami przy jednoczesnym zachowaniu modularności konstrukcji dużego systemu jest bardzo trudne i jest tematem wielu aktualnie prowadzonych badań<sup>52</sup>.

### Ćwiczenie 2.81

Ludwik Myślicielak zauważył, że procedura `apply-generic` może starać się rzutować jeden argument na typ drugiego argumentu nawet wtedy, gdy są one tego samego typu. Dlatego też uważa on, że należy wstawić do tablicy rzutowania procedury „rzutujące” każdy z typów na siebie samego. Oprócz przedstawionej wcześniej procedury `scheme-number->complex` powinniśmy na przykład zainstalować następujące procedury:

```
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
(put-coercion 'scheme-number 'scheme-number
              scheme-number->scheme-number)
(put-coercion 'complex 'complex complex->complex)
```

(a) Co się stanie, jeśli po zainstalowaniu procedur rzutujących Ludwika wywołamy procedurę `apply-generic` z dwoma argumentami typu `scheme-number` lub z dwoma argumentami typu `complex` dla operacji, która nie występuje w tablicy dla tego typu? Założymy na przykład, że zdefiniowaliśmy ogólną operację potęgowania:

```
(define (exp x y) (apply-generic 'exp x y))
```

oraz dodaliśmy do pakietu liczb języka Scheme, ale nie do żadnego innego pakietu, procedurę potęgowania:

<sup>52</sup> Stwierdzenie to jest równie prawdziwe dzisiaj, jak było dwanaście lat temu, gdy pojawiło się w pierwszym wydaniu niniejszej książki. Rozwijanie użytecznych, ogólnych ram, w których można wyrażać relacje między różnymi rodzajami bytów (to, co filozofowie nazywają „ontologią”) okazuje się niezmiennie trudne. Główna różnica między zamętem przed dziesięciu lat a panującym dzisiaj polega na tym, że najrozmaitsze nieadekwatne teorie ontologiczne urzęczystwiono w mnóstwie równie nieadekwatnych języków programowania. Na przykład duża część złożoności obiektowych języków programowania — oraz subtelne i trudne do zrozumienia różnice między współczesnymi językami obiektowymi — koncentruje się wokół podejścia do operacji ogólnych na wzajemnie powiązanych typach. Nasze przedstawienie obiektów obliczeniowych w rozdziale 3 całkowicie pomija te kwestie. Czytelnicy zaznajomieni z programowaniem obiektowym z pewnością zauważą, że w rozdziale 3 mamy dużo do powiedzenia o stanach lokalnych, ale nawet nie wspominamy o „kласach” ani „dziedziczeniu”. Prawdę powiedziawszy, wydaje się nam, że nie można właściwie zająć się tymi problemami jedynie w kategoriach konstrukcji języków programowania, nie sięgając przy tym do prac dotyczących reprezentacji wiedzy i automatycznego wnioskowania.

*;; następujący fragment dodano do pakietu liczb języka Scheme*

*(put 'exp '(scheme-number scheme-number)*

*(lambda (x y) (tag (expt x y)))) ; za pomocą procedury pierwotnej expt*

Co się stanie, jeśli wywołamy procedurę `exp` z dwoma liczbami zespolonymi jako argumentami?

(b) Czy Ludwik ma rację, mówiąc, że coś trzeba zrobić z rzutowaniem argumentów tego samego typu, czy też procedura `apply-generic`, taka jaką jest, działa poprawnie?

(c) Zmień procedurę `apply-generic` tak, aby nie próbowała rzutować argumentów, jeśli oba są tego samego typu.

### Ćwiczenie 2.82

Pokaż, jak uogólnić procedurę `apply-generic`, aby obsługiwała rzutowanie typów dla ogólnego przypadku procedur wieloargumentowych. Jedna z możliwych strategii polega na próbowaniu rzutowania wszystkich argumentów na typ pierwszego argumentu, następnie na typ drugiego argumentu itd. Podaj przykład sytuacji, w której taka strategia (podobnie jak przedstawiona powyżej jej dwuargumentowa wersja) okaże się niedostatecznie ogólna. (Wskazówka: Rozważ przypadek, gdy w tablicy znajdują się odpowiednie operacje na argumentach różnych typów, ale nie zostaną one wywołane).

### Ćwiczenie 2.83

Przypuśćmy, że konstruujesz ogólny system arytmetyczny służący do operowania na wieży typów, pokazanej na rys. 2.25: czyli na liczbach całkowitych, wymiernych, rzeczywistych i zespolonych. Dla każdego z tych typów (z wyjątkiem liczb zespolonych) napisz procedurę rzutującą obiekty tego typu o jeden poziom w górę wieży. Pokaż, jak zainstalować operację ogólną `raise` działającą dla każdego z typów (z wyjątkiem liczb zespolonych).

### Ćwiczenie 2.84

Korzystając z operacji `raise` z ćwiczenia 2.83, zmodyfikuj procedurę `apply-generic` tak, aby rzutowała swoje argumenty na wspólny typ metodą kolejnego ich rzutowania w górę, tak jak omówiliśmy to w niniejszym punkcie. Będziesz musiał wymyślić metodę rozstrzygania, który z dwóch typów znajduje się wyżej w wieży. Zrób to w sposób „kompatybilny” z resztą systemu, tzn. tak, aby nie powstały żadne problemy, gdy zechcemy dodać nowy poziom w wieży.

### Ćwiczenie 2.85

W niniejszym punkcie wspomnialiśmy o metodzie „upraszczania” obiektów danych przez rzutowanie ich jak najniżej w wieży typów. Napisz procedurę `drop` realizującą takie rzutowanie dla wieży opisanej w ćwiczeniu 2.83. Kluczem jest rozstrzygnięcie, w ogólny sposób, czy obiekt może być rzutowany w dół. Na przykład liczba zespolona  $1.5 + 0i$  może być rzutowana w dół do poziomu liczb rzeczywistych (`real`), liczba zespolona  $1 + 0i$  może być rzutowana w dół do poziomu liczb całkowitych (`integer`), a liczba zespolona  $2 + 3i$  nie może być w ogóle rzutowana w dół. Oto sposób, jak można rozstrzygnąć, czy obiekt może być rzutowany w dół. Na początek definiujemy operację ogólną `project` „spychającą” obiekt w dół wieży. Zastosowanie tej operacji

na przykład do liczby zespolonej spowoduje odrzucenie jej części urojonej. Wówczas liczba może być rzutowana w dół, jeśli zastosowanie do niej operacji `project`, a następnie rzutowanie wyniku w górę do wyjściowego typu, za pomocą `raise`, da nam tę samą liczbę. Pokaż w szczegółach, jak zaimplementować ten pomysł, i napisz procedurę `drop` rzutującą obiekt jak najniżej. Będziesz musiał skonstruować różne operacje rzutujące w dół<sup>53</sup> i zainstalować `project` jako operację ogólną w systemie. Będziesz również musiał skorzystać z ogólnego predykatu równości, takiego jak opisany w Ćwiczeniu 2.79. Na koniec, używając `drop`, zmień procedurę `apply-generic` z Ćwiczenia 2.84 tak, aby „upraszczała” swoje wyniki.

### Ćwiczenie 2.86

Przypuśćmy, że chcemy posługiwać się liczbami zespolonymi, których części rzeczywiste, części urojone, wartości bezwzględne i kąty mogą być albo zwykłymi liczbami, albo liczbami wymiernymi, albo innymi liczbami, jakie możemy zechcieć dodać do systemu. Opisz i zaimplementuj zmiany w systemie potrzebne do wprowadzenia powyższych modyfikacji. Będziesz musiał zdefiniować takie operacje ogólne, jak `sine` i `cosine`, dla zwykłych liczb i liczb wymiernych.

### 2.5.3. Przykład: algebra symboliczna

Operowanie symbolicznymi wyrażeniami algebraicznymi to złożony proces ilustrujący wiele z najtrudniejszych problemów pojawiających się w trakcie tworzenia systemów o dużych możliwościach. Ogólnie mówiąc, wyrażenie algebraiczne możemy sobie wyobrazić jako strukturę hierarchiczną — drzewo zbudowane z operatorów i argumentów. Wyrażenia algebraiczne możemy tworzyć, zaczynając od obiektów pierwotnych, takich jak stałe i zmienne, a następnie łącząc je za pomocą operatorów algebraicznych, takich jak dodawanie i mnożenie. Tak jak w innych językach, budujemy abstrakcje umożliwiające oznaczanie w prosty sposób obiektów złożonych. Typowymi abstrakcjami w algebrze symbolicznej są takie pojęcia, jak kombinacja liniowa, wielomian, funkcja wymierna lub funkcja trygonometryczna. Możemy je uważać za „typy” złożone, przydatne przy kierowaniu przetwarzaniem wyrażeń. Moglibyśmy na przykład przedstawić wyrażenie

$$x^2 \sin(y^2 + 1) + x \cos 2y + \cos(y^3 - 2y^2)$$

jako wielomian zmiennej  $x$ , którego współczynniki są wartościami funkcji trygonometrycznych zastosowanych do wielomianów zmiennej  $y$  o współczynnikach całkowitych.

Nie będziemy tutaj próbować opracować pełnego systemu operacji algebraicznych. Takie systemy są niezwykle złożonymi programami zawierającymi eleganckie algorytmy i głęboką wiedzę o algebrze. Przyjrzymy się natomiast

<sup>53</sup> Liczby rzeczywiste można rzutować na całkowite za pomocą procedury pierwotnej `round`, której wynikiem jest liczba całkowita najbliższa argumentowi.

ważnej części operacji algebraicznych: arytmetyce wielomianów. Zilustrujemy rodzaje decyzji, przed jakimi staje konstruktor takiego systemu, i pokażemy, jak może w tym pomóc stosowanie techniki abstrakcji danych i operacji ogólnych.

### Arytmetyka wielomianów

Naszym pierwszym zadaniem, w trakcie konstruowania systemu wykonującego operacje arytmetyczne na wielomianach, jest podjęcie decyzji, czym właściwie jest wielomian. Zwykle wielomiany definiuje się w odniesieniu do pewnych zmiennych (zmiennych *nieoznaczonych* wielomianu). Dla uproszczenia ograniczymy się do wielomianów mających tylko jedną zmienną (czyli *wielomianów jednej zmiennej*)<sup>54</sup>. Zdefiniujemy wielomian jako sumę wyrazów, z których każdy jest albo współczynnikiem, albo potęgą zmiennej nieoznaczonej, albo iloczynem współczynnika i potęgi zmiennej nieoznaczonej. Współczynnik jest zdefiniowany jako wyrażenie algebraiczne, które nie zależy od zmiennej nieoznaczonej wielomianu. Na przykład

$$5x^2 + 3x + 7$$

jest prostym wielomianem zmiennej  $x$ , zaś

$$(y^2 + 1)x^3 + (2y)x + 1$$

jest wielomianem zmiennej  $x$ , którego współczynniki są wielomianami zmiennej  $y$ .

Już w tym momencie pominęliśmy kilka trudnych kwestii. Czy pierwszy z tych wielomianów to ten sam wielomian co  $5y^2 + 3y + 7$ , czy też nie? Rozsądną odpowiedź mogłaby brzmieć: „tak, jeśli rozpatrujemy wielomiany wyłącznie jako funkcje matematyczne, ale nie, jeśli rozpatrujemy wielomiany jako twory składniowe”. Drugi wielomian jest algebraicznie równoważny wielomianowi zmiennej  $y$ , którego współczynniki są wielomianami zmiennej  $x$ . Czy nasz system powinien rozpoznawać takie sytuacje, czy też nie? Ponadto istnieją inne sposoby reprezentowania wielomianów — na przykład w postaci iloczynu czynników lub (w przypadku wielomianów jednej zmiennej) w postaci zbioru pierwiastków, lub przez podanie wartości wielomianu w określonych punktach<sup>55</sup>. Możemy zgrabnie rozstrzygnąć te kwestie, przyjmując, że w naszym

<sup>54</sup> Z drugiej strony, zajmiemy się wielomianami, których współczynniki są wielomianami innych zmiennych. Da nam to, w gruncie rzeczy, tę samą siłę wyrazu co wielomiany wielu zmiennych, choć — jak zobaczymy dalej — będzie to prowadzić do problemów z rzutowaniem typów.

<sup>55</sup> W przypadku wielomianów jednej zmiennej podanie wartości wielomianu w określonych punktach może być szczególnie dogodną reprezentacją. Niezwykle upraszcza ona arytmetykę wielomianów. Chcąc w takiej reprezentacji na przykład dodać do siebie dwa wielomiany, wystarczy dodać ich wartości w odpowiednich punktach. Chcąc przekształcić wielomian z powrotem do tradycyjnej postaci, możemy zastosować wzór interpolacyjny Lagrange'a, który pozwala określić współczynniki wielomianu stopnia  $n$  na podstawie wartości wielomianu w  $n + 1$  punktach.

systemie operacji algebraicznych „wielomian” będzie konkretnym tworem składowym, a nie kryjącym się za nim pojęciem matematycznym.

Musimy się teraz zastanowić, jak się zabrać za wykonywanie operacji arytmetycznych na wielomianach. W naszym prostym systemie będziemy rozważać tylko dodawanie i mnożenie wielomianów. Co więcej, będziemy wymagać, aby dwa wielomiany, na których wykonujemy operację, były wielomianami tej samej zmiennej.

Zabierzemy się do konstruowania naszego systemu, postępując zgodnie ze znaną nam zasadą abstrakcji danych. Wielomiany będziemy reprezentować za pomocą struktury danych o nazwie *poly*, składającej się ze zmiennej nieoznaczonej i kolekcji wyrazów wielomianu. Zakładamy, że są dostępne selektory *variable* i *term-list*, które wydobywają te dwie części ze struktury *poly*, oraz konstruktor *make-poly* tworzący na podstawie zmiennej i listy wyrazów strukturę *poly*. Zmienna będzie po prostu symbolem, co pozwoli nam porównywać zmienne za pomocą procedury *same-variable?* z punktu 2.3.2. Następujące procedury definiują dodawanie i mnożenie struktur *poly*:

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                  (add-terms (term-list p1)
                             (term-list p2)))
      (error "Poly zawierają różne zmienne -- ADD-POLY"
            (list p1 p2)))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                  (mul-terms (term-list p1)
                             (term-list p2)))
      (error "Poly zawierają różne zmienne -- MUL-POLY"
            (list p1 p2))))
```

Aby dołączyć wielomiany do naszego ogólnego systemu arytmetycznego, musimy je wyposażyć w znaczniki typów. Użyjemy znacznika *polynomial* i zainstalujemy odpowiednie operacje na wielomianach ze znacznikami w tablicy operacji i typów. Cały nasz kod zawsze w procedurze instalującej pakiet wielomianów, podobnej do tej z punktu 2.5.1:

```
(define (install-polynomial-package)
  ;; procedury wewnętrzne
  ;; reprezentacja poly
  (define (make-poly variable term-list)
    (cons variable term-list))
  (define (variable p) (car p))
```

```

(define (term-list p) (cdr p))
⟨procedury same-variable? i variable? z punktu 2.3.2⟩

;; reprezentacja wyrazów wielomianów i ich list
⟨procedury adjoin-term ... coeff przedstawione poniżej⟩
(define (add-poly p1 p2) ...)
⟨procedury używane przez add-poly⟩
(define (mul-poly p1 p2) ...)
⟨procedury używane przez mul-poly⟩

;; interfejs dla reszty systemu
(define (tag p) (attach-tag 'polynomial p))
(put 'add '(polynomial polynomial)
      (lambda (p1 p2) (tag (add-poly p1 p2))))
(put 'mul '(polynomial polynomial)
      (lambda (p1 p2) (tag (mul-poly p1 p2))))
(put 'make 'polynomial
      (lambda (var terms) (tag (make-poly var terms))))
'done)

```

Dodając wielomiany, dodajemy do siebie wyrazy tego samego stopnia (tzn. o tej samej potędze zmiennej nieoznaczonej). Tworzymy przy tym nowy wyraz, tego samego stopnia, którego współczynnik jest sumą współczynników dodawanych wyrazów. Wyrazy jednego składnika, dla których brak wyrazów tego samego stopnia w drugim składniku, są po prostu gromadzone w tworzonej sumie wielomianów.

Chcąc operować listami wyrazów, zakładamy, że są dostępne konstruktory: `the-empty-termlist`, którego wynikiem jest pusta lista wyrazów, oraz `adjoin-term`, dołączający nowy wyraz do listy wyrazów. Zakładamy, że są również dostępne: predykat `empty-termlist?` określający, czy dana lista wyrazów jest pusta, selektor `first-term` wydobywający z listy wyrazów wyraz najwyższego stopnia oraz selektor `rest-terms`, którego wynikiem są wszystkie wyrazy z wyjątkiem wyrazu najwyższego stopnia. Chcąc zaś operować na wyrazach, zakładamy, że są dostępne: konstruktor `make-term`, który tworzy wyraz o zadanym stopniu i współczynniku, oraz selektory `order` i `coeff`, których wynikami są, odpowiednio, stopień i współczynnik wyrazu. Operacje te pozwalają nam traktować zarówno wyrazy, jak i listy wyrazów jako dane abstrakcyjne, których konkretnymi reprezentacjami możemy się zajmować niezależnie.

Oto procedura tworząca listę wyrazów sumy dwóch wielomianów<sup>56</sup>:

<sup>56</sup> Operacja ta przypomina bardzo operację `union-set` dla zbiorów reprezentowanych jako listy uporządkowane, z ćwiczenia 2.62. W rzeczywistości, jeśli wyobrażymy sobie listę wyrazów wielomianu jako zbiór uporządkowany ze względu na stopnie wyrazów, to program obliczający listę wyrazów sumy jest prawie identyczny z `union-set`.

```
(define (add-terms L1 L2)
  (cond ((empty-term-list? L1) L2)
        ((empty-term-list? L2) L1)
        (else
          (let ((t1 (first-term L1)) (t2 (first-term L2)))
            (cond ((> (order t1) (order t2))
                   (adjoin-term
                     t1 (add-terms (rest-terms L1) L2)))
                  ((< (order t1) (order t2))
                   (adjoin-term
                     t2 (add-terms L1 (rest-terms L2))))
                  (else
                    (adjoin-term
                      (make-term (order t1)
                                 (add (coeff t1) (coeff t2)))
                      (add-terms (rest-terms L1)
                                 (rest-terms L2))))))))))
```

Ważnym szczegółem, na który trzeba zwrócić uwagę, jest to, że do sumowania współczynników użyliśmy tutaj ogólnej procedury dodawania add. Jak zobaczymy dalej, będzie to miało istotne znaczenie.

Chcąc pomnożyć przez siebie dwie listy wyrazów, mnożymy każdy wyraz z pierwszej listy przez wszystkie wyrazy drugiej listy, wielokrotnie używając procedury mul-term-by-all-terms, która mnoży dany wyraz przez wszystkie wyrazy z danej listy. Powstające w ten sposób listy (po jednej dla każdego wyrazu z pierwszej listy) są sumowane. Mnożąc dwa wyrazy, tworzymy wyraz, którego stopień jest sumą stopni czynników i którego współczynnik jest iloczynem współczynników czynników:

```
(define (mul-terms L1 L2)
  (if (empty-term-list? L1)
      (the-empty-term-list)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2)))))

(define (mul-term-by-all-terms t1 L)
  (if (empty-term-list? L)
      (the-empty-term-list)
      (let ((t2 (first-term L)))
        (adjoin-term
          (make-term (+ (order t1) (order t2))
                     (mul (coeff t1) (coeff t2)))
          (mul-term-by-all-terms t1 (rest-terms L))))))
```

W zasadzie jest to całe dodawanie i mnożenie wielomianów. Zauważmy, że ponieważ operujemy na wyrazach wielomianów, używając procedur ogólnych

add i mul, nasz pakiet wielomianów automatycznie potrafi obsłużyć współczynniki dowolnego typu znanego w ogólnym pakiecie arytmetycznym. Jeśli dodamy mechanizm rzutowania typów, taki jak ten omówiony w punkcie 2.5.2, to automatycznie będziemy również mogli wykonywać operacje na wielomianach o współczynnikach różnych typów, takich jak

$$\left[3x^2 + (2 + 3i)x + 7\right] \cdot \left[x^4 + \frac{2}{3}x^2 + (5 + 3i)\right]$$

Ponieważ procedury dodawania i mnożenia wielomianów, add-poly i mul-poly, zainstalowaliśmy w ogólnym systemie arytmetycznym jako operacje add i mul dla typu polynomial, więc nasz system automatycznie potrafi również wykonywać takie operacje, jak

$$\left[(y + 1)x^2 + (y^2 + 1)x + (y - 1)\right] \cdot \left[(y - 2)x + (y^3 + 7)\right]$$

Wynika to stąd, że gdy system próbuje wykonać operację na współczynnikach, przekierowuje ją do add i mul. Ponieważ współczynniki same są wielomianami (zmiennej  $y$ ), zostaną na nich wykonane operacje add-poly i mul-poly. W rezultacie otrzymujemy rodzaj rekursji „sterowanej danymi”, w której na przykład wywołanie mul-poly spowoduje rekurencyjne wywołanie tej samej procedury w celu pomnożenia współczynników. Jeśli współczynniki współczynników same są wielomianami (co może być stosowane do reprezentowania wielomianów trzech zmiennych), to dzięki sterowaniu danymi system zagłębi się w kolejny poziom rekursji — itd. przez tyle poziomów, ile wymaga struktura danych<sup>57</sup>.

### Reprezentowanie list wyrazów

Musimy w końcu stawić czoło zadaniu zaimplementowania dobrej reprezentacji list wyrazów wielomianów. W rzeczywistości lista wyrazów jest zbiorem współczynników indeksowanych stopniami wyrazów. Tak więc dowolna z metod reprezentowania zbiorów, omówionych w punkcie 2.3.3, może być do tego zastosowana. Z kolei nasze procedury add-terms i mul-terms zawsze przeglądają listy wyrazów sekwencyjnie — od najwyższych do najniższych stopni. Dlatego też zastosujemy pewien rodzaj reprezentacji za pomocą list uporządkowanych.

<sup>57</sup> Aby wszystko szło gładko, powinniśmy również dodać do naszego ogólnego systemu arytmetycznego możliwość rzutowania „liczby” na wielomian, traktując ją jak wielomian stopnia zero, którego współczynnik jest daną liczbą. Jest to konieczne, jeśli chcemy wykonywać takie operacje, jak

$$\left[x^2 + (y + 1)x + 5\right] + \left[x^2 + 2x + 1\right]$$

która wymaga dodania współczynników  $y + 1$  i  $2$ .

Jaką strukturę powinna mieć lista reprezentująca listę wyrazów wielomianu? Jeden z czynników — jakie powinniśmy rozważyć — to, jak „gęste” są wielomiany, którymi chcemy operować. Mówimy, że wielomian jest *gęsty* (ang. *dense*), gdy większość jego współczynników jest różna od zera. Jeśli zaś wiele jego współczynników (wyrazów) ma wartość zero, to mówimy, że jest on *rzadki* (ang. *sparse*). Na przykład wielomian

$$A : x^5 + 2x^4 + 3x^2 - 2x - 5$$

jest gęsty, natomiast wielomian

$$B : x^{100} + 2x^2 + 1$$

jest rzadki.

Listy wyrazów wielomianów gęstych najefektywniej reprezentuje się jako listy współczynników. Na przykład podany powyżej wielomian *A* można by dobrze reprezentować jako (1 2 0 3 -2 -5). Stopień wyrazu w takiej reprezentacji jest równy długości zaczynającej się od niego podlisty minus jeden<sup>58</sup>. Byłaby to jednak okropna reprezentacja wielomianów rzadkich, takich jak *B* — byłaby to olbrzymia lista złożona z zer przerywanych tylko gdzieniegdzie niezerowymi współczynnikami. Rozsądniejszą reprezentacją listy wyrazów wielomianów rzadkich jest lista zawierająca tylko niezerowe wyrazy, gdzie każdy wyraz jest reprezentowany przez listę złożoną z jego stopnia i współczynnika. Przy takim schemacie reprezentacji wielomian *B* jest efektywnie reprezentowany jako ((100 1) (2 2) (0 1)). Ze względu na to, że większość operacji jest wykonywana na wielomianach rzadkich, przyjmiemy taką właśnie reprezentację. Zakładamy przy tym, że wyrazy wielomianu są uporządkowane na liście od najwyższego do najniższego stopnia. Po podjęciu tych decyzji implementacja selektorów i konstruktorów wyrazów wielomianów i ich list jest całkiem prosta<sup>59</sup>:

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
```

<sup>58</sup> W tych przykładach wielomianów zakładamy, że ogólny system arytmetyczny został zaimplementowany z użyciem wewnętrznego mechanizmu kontroli typów, jak to zasugerowaliśmy w ćwiczeniu 2.78. W ten sposób współczynniki będące zwykłymi liczbami mogą same siebie reprezentować, a nie być reprezentowane przez pary, których car jest symbolem `scheme-number`.

<sup>59</sup> Chociaż zakładamy, że listy wyrazów są uporządkowane, jednak zaimplementowaliśmy operację `adjoin-term` jako zwykły `cons` dodający nowy wyraz do istniejącej listy. Możemy tak zrobić, o ile mamy pewność, że procedury używające `adjoin-term` (takie jak `add-terms`) zawsze wywołują tę procedurę, wstawiając wyraz wyższego stopnia niż wszystkie wyrazy znajdujące się na liście. Gdybyśmy chcieli uniknąć takiego założenia, to moglibyśmy zaimplementować `adjoin-term` podobnie do konstruktora `adjoin-set` dla reprezentacji zbiorów za pomocą list uporządkowanych (zob. ćwiczenie 2.61).

---

```
(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

gdzie `=zero?` jest takie jak zdefiniowane w ćwiczeniu 2.80. (Zobacz również ćwiczenie 2.87 poniżej).

Użytkownicy pakietu wielomianów mogą tworzyć wielomiany (ze znacznikami) za pomocą procedury

```
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

### Ćwiczenie 2.87

Zainstaluj w ogólnym pakiecie arytmetycznym predykat `=zero?` określony dla wielomianów. Pozwoli to procedurze `adjoin-term` działać dla wielomianów, których współczynniki same są wielomianami.

### Ćwiczenie 2.88

Rozszerz system wielomianów o operację odejmowania. (Wskazówka: Może się okazać przydatne zdefiniowanie ogólnej operacji zmiany znaku na przeciwny).

### Ćwiczenie 2.89

Zdefiniuj procedury implementujące opisaną powyżej reprezentację list wyrazów właściwą dla wielomianów gęstych.

### Ćwiczenie 2.90

Przypuśćmy, że chcemy dysponować systemem wielomianów, który jest efektywny zarówno dla wielomianów rzadkich, jak i wielomianów gęstych. Można to osiągnąć, dopuszczając istnienie w systemie dwóch rodzajów reprezentacji list wyrazów wielomianów. Jest to sytuacja analogiczna do przykładu liczb zespolonych z podrozdziału 2.4, gdzie dopuściliśmy istnienie zarówno reprezentacji prostokątnej, jak i reprezentacji biegunowej. Chcąc tak zrobić, musimy rozróżnić dwa rodzaje list wyrazów wielomianów i uogólnić operacje na listach wyrazów. Wprowadź do systemu wielomianów zmiany implementujące takie uogólnienie. Wymaga to większego wysiłku, a nie tylko lokalnej zmiany.

### Ćwiczenie 2.91

Wielomiany jednej zmiennej można dzielić przez siebie, otrzymując iloraz i resztę z dzielenia, będące też wielomianami. Na przykład:

$$\frac{x^5 - 1}{x^2 - 1} = x^3 + x, \text{ reszta } x - 1$$

Dzielenie takie możemy wykonywać tak jak dzielenie w słupku. Oznacza to, że dzielimy wyraz dzielnej o najwyższym stopniu przez wyraz dzielnika o najwyższym stopniu. Wynik jest pierwszym wyrazem ilorazu. Następnie mnożymy uzyskany wyraz przez dzielnik i odejmujemy wynik od dzielnej. Pozostałe wyrazy ilorazu obliczamy, dzieląc rekurencyjnie otrzymaną różnicę przez dzielnik. Przerywamy ten proces, gdy stopień dzielnika jest większy niż stopień dzielnej — wówczas dzielna jest równa reszcie z dzielenia. Również jeżeli dzielna jest kiedykolwiek równa zeru, to iloraz i reszta z dzielenia są równe zeru.

Wzorując się na procedurach `add-poly` i `mul-poly`, możemy skonstruować procedurę `div-poly`. Procedura ta sprawdza, czy dwa wielomiany są wielomianami tej samej zmiennej. Jeśli tak, `div-poly` odrywa zmienną i przekazuje zadanie procedurze `div-terms`, która wykonuje dzielenie na listach wyrazów wielomianów. Na koniec `div-terms` dodaje do wyniku `div-terms` zmienną nieoznaczoną wielomianu. Wygodnie jest, aby procedura `div-terms` obliczała zarówno iloraz, jak i resztę z dzielenia. Argumentami `div-terms` mogą być dwie listy wyrazów, a wynikiem może być lista złożona z listy wyrazów ilorazu oraz listy wyrazów reszty.

Uzupełnij następującą definicję procedury `div-terms`, wpisując brakujące wyrażenia. Użyj jej do zaimplementowania `div-poly`, której argumentami są dwie struktury `poly`, a wynikiem jest lista złożona ze struktur `poly` iloczynu i reszty.

```
(define (div-terms L1 L2)
  (if (empty-term-list? L1)
      (list (the-empty-term-list) (the-empty-term-list))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-term-list) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     (oblicz rekurencyjnie pozostałą część wyniku)
                     )))
                (utwórz pełny wynik)
                )))))
  ))))))
```

### Hierarchie typów w algebrze symbolicznej

Nasz system wielomianów ilustruje, jak obiekty jednego typu (wielomiany) mogą faktycznie być obiektami złożonymi, których składowymi jest wiele obiektów różnych typów. Nie stanowi to prawdziwego problemu w definiowaniu operacji ogólnych. Musimy tylko zainstalować odpowiednie operacje ogólne wykonujące niezbędne czynności na składowych typów złożonych. W rzeczywistości zobaczyliśmy, że wielomiany tworzą rodzaj „rekurencyjnej abstrakcji danych” w ten sposób, że składowe wielomianów mogą same być wielomianami. Nasze operacje ogólne i styl programowania sterowanego danymi radzą sobie z tym bez większego kłopotu.

Jednakże algebra wielomianów jest systemem, w którym typy danych nie tworzą w naturalny sposób wieży. Można na przykład mieć wielomiany zmien-

nej  $x$ , których współczynniki są wielomianami zmiennej  $y$ . Można również mieć wielomiany zmiennej  $y$ , których współczynniki są wielomianami zmiennej  $x$ . Żaden z tych typów nie znajduje się w naturalny sposób „powyżej” drugiego, a mimo to często trzeba dodawać do siebie elementy takich typów. Można to zrobić na kilka sposobów. Jeden z nich polega na przekształceniu jednego wielomianu do typu drugiego wielomianu, przez rozwinięcie i przegrupowanie jego wyrazów tak, aby oba wielomiany miały te same główne zmienne nieoznaczone. Strukturze typów można nadać kształt wieży, porządkując zmienne i przekształcając zawsze każdy wielomian do „postaci kanonicznej”, w której dominuje zmienna o najwyższym priorytecie, a pozostałe zmienne są ukryte we współczynnikach. Strategia ta funkcjonuje całkiem dobrze poza tym, że przekształcane wielomiany mogą się niepotrzebnie rozrastać, przez co ciężej może być je przeczytać, a operacje na nich mogą być mniej efektywne. Strategia wieży najwyższej nie pasuje do tej dziedziny, jak i do żadnej dziedziny, w której użytkownik może dynamicznie wprowadzać nowe typy, łącząc na najrozmaitsze sposoby stare typy, jak na przykład za pomocą funkcji trygonometrycznych, szeregów potęgowych czy całek.

Nie powinno nas dziwić, że kontrolowanie rzutowania typów stanowi poważny problem przy konstruowaniu systemów operacji algebraicznych o dużych możliwościach. Duża część złożoności takich systemów wiąże się z zależnościami między różnorodnymi typami. Możemy wręcz powiedzieć, że nie rozumiemy jeszcze do końca rzutowania typów. Tak naprawdę nie rozumiemy do końca pojęcia typu danych. Niemniej to, co wiemy, daje nam do ręki potężne zasady strukturalności i modularności, pomagające w tworzeniu dużych systemów.

### Ćwiczenie 2.92

Rozszerz pakiet wielomianów, narzucając pewne uporządkowanie zmiennych, tak aby dodawanie i mnożenie wielomianów działało dla wielomianów różnych zmiennych. (To nie jest łatwe!)

### Rozszerzone ćwiczenie: funkcje wymierne

Możemy rozszerzyć nasz ogólny system arytmetyczny o *funkcje wymierne*. Są to „ułamki”, których liczniki i mianowniki są wielomianami, jak na przykład

$$\frac{x+1}{x^3-1}$$

System ten powinien umieć dodawać, odejmować, mnożyć i dzielić funkcje wymierne, a także wykonywać takie operacje jak

$$\frac{x+1}{x^3-1} + \frac{x}{x^2-1} = \frac{x^3+2x^2+3x+1}{x^4+x^3-x-1}$$

(Powyższą sumę uproszczono przez skrócenie wspólnych czynników. Zwykłe pomnożenie „na krzyż” liczników i mianowników dałoby w wyniku wielomian czwartego stopnia podzielony przez wielomian piątego stopnia).

Jeżeli tak zmodyfikujemy nasz pakiet arytmetyki liczb wymiernych, aby używał operacji ogólnych, to będzie on robił wszystko, co trzeba, z wyjątkiem skracania ułamków.

### Ćwiczenie 2.93

Zmodyfikuj pakiet arytmetyki liczb wymiernych, aby używał operacji ogólnych, ale procedurę `make-rat` zmień tak, żeby nie starała się skracać ułamków. Przetestuj swój program, tworząc funkcję wymierną za pomocą wywołania procedury `make-rational` dla dwóch argumentów będących wielomianami:

```
(define p1 (make-polynomial 'x '((2 1)(0 1))))
(define p2 (make-polynomial 'x '((3 1)(0 1))))
(define rf (make-rational p2 p1))
```

Następnie dodaj `rf` do siebie za pomocą `add`. Zauważ, że procedura dodawania nie skracą ułamków.

Ułamki złożone z wielomianów możemy skracać w ten sam sposób co w przypadku liczb całkowitych: zmodyfikujmy `make-rat` tak, aby dzielił zarówno licznik, jak i mianownik przez ich największy wspólny dzielnik (NWD). Pojęcie „największego wspólnego dzielnika” ma sens również w przypadku wielomianów. W rzeczywistości możemy obliczyć NWD dwóch wielomianów, używając w zasadzie takiego samego algorytmu Euklidesa jak dla liczb całkowitych<sup>60</sup>. Jego wersja dla liczb całkowitych jest następująca:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Możemy jej użyć, wprowadzając pewne oczywiste zmiany do operacji obliczającej NWD, tak aby działała ona dla list wyrazów wielomianów:

<sup>60</sup> Fakt, że algorytm Euklidesa działa dla wielomianów, można w algebraze wyrazić formalnie, mówiąc, że wielomiany tworzą *pierścień euklidesowy*. Pierścień euklidesowy to taki zbiór, na którym są określone operacje dodawania, odejmowania oraz przemienneego mnożenia, a także dla każdego jego elementu  $x$  różnego od zera jest określona dodatnia i całkowita „miara”  $m(x)$  spełniająca następujące warunki: dla dowolnych  $x$  i  $y$  różnych od zera  $m(xy) \geq m(x)$  oraz dla dowolnego  $x$  różnego od zera i dowolnego  $y$  istnieją takie  $q$  i  $r$ , że  $y = qx + r$  i albo  $r = 0$ , albo  $m(r) < m(x)$ . Z abstrakcyjnego punktu widzenia jest to dokładnie to, co jest potrzebne do udowodnienia poprawności algorytmu Euklidesa. W przypadku liczb całkowitych miara liczby jest jej wartość bezwzględna. W przypadku wielomianów miara wielomianu jest jego stopień.

```
(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (remainder-terms a b))))
```

gdzie `remainder-terms` wyłuskuje z listy będącej wynikiem operacji dzielenia list wyrazów `div-terms` (zaimplementowanej w ćwiczeniu 2.91) resztę z dzielenia.

### Ćwiczenie 2.94

Korzystając z `div-terms`, zaimplementuj procedurę `remainder-terms` i zastosuj ją do zdefiniowania `gcd-terms` w przedstawiony powyżej sposób. Następnie napisz procedurę `gcd-poly` obliczającą NWD dwóch wielomianów reprezentowanych przez dwie struktury `poly`. (Procedura ta powinna zgłaszać błąd, jeśli jej argumenty nie są wielomianami tej samej zmiennej). Zainstaluj w systemie operację ogólną `greatest-common-divisor`, która dla wielomianów sprowadza się do `gcd-poly`, a dla zwykłych liczb do zwykłego `gcd`. Przetestuj ją na następujących danych:

```
(define p1 (make-polynomial 'x '((4 1) (3 -1) (2 -2) (1 2))))
(define p2 (make-polynomial 'x '((3 1) (1 -1))))
(greatest-common-divisor p1 p2)
```

i sprawdź ręcznie uzyskane wyniki.

### Ćwiczenie 2.95

Zdefiniuj następujące wielomiany:

$$P_1 : x^2 - 2x + 1$$

$$P_2 : 11x^2 + 7$$

$$P_3 : 13x + 5$$

Następnie zdefiniuj  $Q_1$  jako iloczyn  $P_1$  i  $P_2$ , a  $Q_2$  jako iloczyn  $P_1$  i  $P_3$ , po czym zastosuj procedurę `greatest-common-divisor` (z ćwiczenia 2.94) do obliczenia NWD  $Q_1$  i  $Q_2$ . Zwrót uwagę, że wynik nie jest taki sam jak  $P_1$ . W tym przykładzie, w obliczeniach pojawiają się wartości niecałkowite, co powoduje trudności z algorytmem NWD<sup>61</sup>. Aby zrozumieć, co się dzieje, spróbuj prześledzić działanie `gcd-terms` w trakcie obliczania NWD lub spróbuj ręcznie wykonać dzielenie.

Możemy rozwiązać problem wskazany w ćwiczeniu 2.95, jeśli w następujący sposób zmodyfikujemy algorytm NWD (co w rzeczywistości pomaga tylko dla wielomianów o współczynnikach całkowitych). Zanim przystąpimy do jakiegokolwiek dzielenia wielomianów w trakcie obliczania NWD, mnożymy dzielną przez stały czynnik całkowity, wybrany tak, aby w trakcie dzielenia

<sup>61</sup> W takich implementacjach jak MIT Scheme wynikiem jest wielomian, który jest wprawdzie dzielnikiem  $Q_1$  i  $Q_2$ , ale o współczynnikach wymiernych. W wielu innych implementacjach języka Scheme, w których wynik dzielenia liczb całkowitych jest liczbą dziesiętną ograniczoną precyzji, możemy nie uzyskać poprawnego dzielnika.

nie pojawiły się żadne ułamki. W ten sposób nasz wynik będzie się różnił od faktycznego NWD o stały czynnik całkowity, nie ma to jednak najmniejszego znaczenia w przypadku upraszczania funkcji wymiernych — zarówno licznik, jak i mianownik zostaną podzielone przez NWD, a ów stały czynnik się skróci.

Mówiąc bardziej precyzyjnie, jeśli  $P$  i  $Q$  są wielomianami, to niech  $O_1$  będzie stopniem  $P$  (tzn. stopniem najwyższego wyrazu  $P$ ), a  $O_2$  niech będzie stopniem  $Q$ . Niech  $c$  będzie współczynnikiem wiodącym wielomianu  $Q$ . Można wówczas pokazać, że jeżeli pomnożymy  $P$  przez *czynnik zachowania współczynników całkowitych*  $c^{1+O_1-O_2}$ , to tak powstały wielomian można podzielić przez  $Q$  za pomocą algorytmu div-terms bez wprowadzania żadnych ułamków. Operacja mnożenia dzielnej przez stałą, a następnie dzielenie jej, jest czasem nazywana *pseudodzieleniem*  $P$  przez  $Q$ . Reszta z takiego pseudodzielenia jest nazywana *pseudoresztą*.

### Ćwiczenie 2.96

- (a) Zaimplementuj procedurę pseudoremainder-terms, taką samą jak remainder-terms z wyjątkiem tego, że przed wywołaniem div-terms mnoży dzielną przez opisany powyżej czynnik zachowania współczynników całkowitych. Zmodyfikuj gcd-terms tak, aby korzystała z pseudoremainder-terms, i sprawdź na przykładzie z ćwiczenia 2.95, że teraz wyniki greatest-common-divisor mają współczynniki całkowite.
- (b) NWD ma teraz współczynniki całkowite, ale są one większe od współczynników wielomianu  $P_1$ . Zmodyfikuj gcd-terms tak, aby usuwała wspólne czynniki ze współczynników wyniku, dzieląc wszystkie współczynniki przez ich największy wspólny dzielnik (będący liczbą całkowitą).

A oto jak można skrócić funkcję wymierną:

- Obliczamy NWD licznika i mianownika, używając wersji gcd-terms z ćwiczenia 2.96.
- Gdy uzyskamy NWD, wówczas przed podzieleniem licznika i mianownika przez NWD mnożymy je przez ten sam czynnik zachowania współczynników całkowitych. W ten sposób w wyniku dzielenia nie powstaną żadne niecałkowite współczynniki. Jako czynnika można użyć współczynnika wiodącego NWD podniesionego do potęgi  $1+O_1-O_2$ , gdzie  $O_2$  jest stopniem NWD, a  $O_1$  to maksimum ze stopni licznika i mianownika. Dzięki temu mamy pewność, że w wyniku dzielenia licznika i mianownika przez NWD nie powstaną żadne ułamki.
- Wynikiem tej operacji będzie licznik i mianownik o współczynnikach całkowitych. Zwykle współczynniki te, ze względu na czynnik zachowania współczynników całkowitych, są bardzo duże. Tak więc ostatni krok polega na usunięciu wszystkich zbędnych czynników, poprzez obliczenie największego

wspólnego dzielnika (będącego liczbą całkowitą) wszystkich współczynników licznika i mianownika i podzielenie ich przez niego.

### Ćwiczenie 2.97

(a) Zaimplementuj ten algorytm w postaci procedury `reduce-terms`, której argumentami są dwie listy `n` i `d` wyrazów wielomianów, a wynikiem jest lista `(nn, dd)`, gdzie `nn` i `dd` powstają przez skrócenie `n` i `d` za pomocą podanego powyżej algorytmu. Napisz również procedurę `reduce-poly`, analogiczną do `add-poly`, która sprawdza, czy dwie struktury `poly` reprezentują wielomiany tej samej zmiennej. Jeśli tak, to procedura ta odrywa zmienną i przekazuje problem do `reduce-terms`, a następnie doczepia zmienną z powrotem do list wyrazów będących wynikiem `reduce-terms`.

(b) Zdefiniuj procedurę analogiczną do `reduce-terms`, która robi to, co pierwotna `make-rat` robiła dla liczb całkowitych:

```
(define (reduce-integers n d)
  (let ((g (gcd n d)))
    (list (/ n g) (/ d g))))
```

i zdefiniuj `reduce` jako operację ogólną, która wywołuje `apply-generic`, aby przekierować obliczenia albo do `reduce-poly` (w przypadku argumentów typu `polynomial`), albo do `reduce-integers` (w przypadku argumentów typu `scheme-number`). Możesz teraz z łatwością sprawić, aby pakiet arytmetyki liczb wymiernych skracał ułamki w ten sposób, że `make-rat` wywołuje `reduce` przed utworzeniem z danego licznika i mianownika liczby wymiernej. Teraz system obsługuje wyrażenia wymierne zbudowane zarówno za pomocą liczb całkowitych, jak i wielomianów. Przetestuj swój program na przykładzie z początku tego rozszerzonego ćwiczenia:

```
(define p1 (make-polynomial 'x '((1 1)(0 1))))
(define p2 (make-polynomial 'x '((3 1)(0 -1))))
(define p3 (make-polynomial 'x '((-1 1))))
(define p4 (make-polynomial 'x '((-2 1)(0 -1)))))

(define rf1 (make-rational p1 p2))
(define rf2 (make-rational p3 p4))

(+ rf1 rf2)
```

Sprawdź, czy uzyskałeś prawidłowo skrócone i poprawne wyniki.

Obliczanie NWD jest nieodłączną częścią każdego systemu wykonującego operacje na funkcjach wymiernych. Algorytm zastosowany powyżej, choć matematycznie prosty, jest niezmiernie wolny. Jego powolność wynika częściowo z dużej liczby dzielen, a częściowo z ogromnej wielkości pośrednich współczynników powstających w trakcie pseudodzielenia. Jednym z aktywnych obszarów, w ramach rozwijania systemów operacji algebraicznych, jest konstruowanie lepszych algorytmów obliczających NWD wielomianów<sup>62</sup>.

<sup>62</sup> Niezwykle efektywna i elegancka metoda obliczania NWD wielomianów została odkryta przez Richarda Zippela [111]. Jest to algorytm probabilistyczny, tak jak przedstawiony w rozdziale 1 szybki test pierwszości. Zippel opisuje tę metodę razem z innymi sposobami obliczania NWD wielomianów w [112].

# 3

## Modularność, obiekty i stany

*Μεταβάλλον ἀναπαύεται*

(Nawet jeśli coś się zmienia, to i tak pozostaje takie samo).

Heraklit

Plus ça change, plus c'est la même chose.

(Im bardziej coś się zmienia, tym bardziej pozostaje takie samo).

Alphonse Karr

W poprzednich rozdziałach przedstawiliśmy podstawowe elementy, z których są zbudowane programy. Zobaczyliśmy, jak można łączyć procedury pierwotne i dane pierwotne w obiekty złożone. Dowiedzieliśmy się, jak istotna jest rola abstrakcji w radzeniu sobie ze złożonością dużych systemów. Jednakże narzędzia te nie są wystarczające do konstruowania programów. Efektywne tworzenie programów wymaga również pewnych zasad organizacyjnych, pomocnych w opracowywaniu ogólnych konstrukcji programów. W szczególności potrzebne nam są strategie pomagające w nadawaniu dużym systemom struktury *modularnej*, tzn. takiej, w której w „naturalny” sposób dzielą się one na spójne fragmenty, które można niezależnie opracowywać i pielęgnować.

Jedna z potężnych strategii projektowych, szczególnie przydatna przy tworzeniu programów modelujących systemy fizyczne, polega na wzorowaniu struktury systemu na strukturze modelowanego systemu. Dla każdego obiektu w systemie tworzymy odpowiadający mu obiekt obliczeniowy. Dla każdego działania (akcji) w systemie definiujemy w modelu obliczeniowym odpowiadającą mu operację symboliczną. Stosując tę strategię, mamy nadzieję, że rozszerzanie systemu o nowe obiekty lub działania nie będzie wymagało strategicznych zmian w programie, a jedynie wprowadzenia symbolicznych odpowiedników nowych obiektów lub akcji. Gdyby udało nam się zastosować taką organizację systemu, to rozszerzając go o nowe możliwości lub usuwając błędy z istniejącego już kodu, moglibyśmy ograniczyć się do lokalnych zmian w określonej części systemu.

Przy takim podejściu sposób organizacji dużego programu w znacznym stopniu zależy od naszego sposobu pojmowania modelowanego systemu.

W niniejszym rozdziale zbadamy dwie znaczące strategie projektowe wywołujące się z dwóch różnych sposobów patrzenia na strukturę systemów w otaczającym nas świecie. Pierwsza z nich koncentruje się na *obiektach*, ukazując duży system jako zbiór różnych obiektów, których zachowania mogą zmieniać się w czasie. Druga zaś koncentruje się na *strumieniach* informacji przepływającej w systemie, przedstawiając system podobnie, jak to czynią inżynierowie elektronicy z systemami przetwarzania sygnałów.

Oba podejścia, obiektowe i przetwarzania strumieni, dotykają istotnych kwestii związanych z językiem programowania. W przypadku podejścia obiektowego musimy się skupić na zmianach zachodzących w obiekcie obliczeniowym, przy jednoczesnym zachowaniu jego tożsamości. Tym samym będziemy zmuszeni do porzucenia naszego starego podstawieniowego modelu obliczeń (punkt 1.1.5) na rzecz bardziej mechanistycznego, choć bardziej skomplikowanego w teoretycznych zastosowaniach, *środowiskowego modelu* (ang. *environment model*) obliczeń. Trudności w operowaniu obiektemi, ich zmiany i ich tożsamość wynikają w zasadniczy sposób z konieczności zmagania się z czasem w naszych modelach obliczeniowych. Trudności te stają się jeszcze większe, gdy dopuścimy możliwość równoległego wykonywania programów. Podejście strumieniowe możemy najlepiej wykorzystać, gdy oddzielimy w naszym modelu symulowany czas od porządku zdarzeń zachodzących w komputerze w trakcie obliczeń. Dokonamy tego za pomocą techniki nazywanej *odraczaniem obliczeń* (ang. *delayed evaluation*).

### 3.1. Przypisanie i stan lokalny

Zazwyczaj postrzegamy świat jako wypełniony niezależnymi obiektami, z których każdy charakteryzuje się stanem zmieniającym się w czasie. Mówimy, że obiekt „ma stan”, gdy jego zachowanie zależy od jego wcześniejszej historii. Konto bankowe na przykład ma stan w tym sensie, że odpowiedź na pytanie „Czy mogę wypłacić 100 dolarów?” zależy od historii wpłat i wypłat. Stan obiektu możemy opisać za pomocą jednej lub więcej *zmiennych stanu* (ang. *state variables*), zawierających w sobie dostateczną ilość informacji z historii obiektu, aby określić jego aktualne zachowanie. W przypadku prostego systemu bankowego możemy opisać stan konta, podając jego bieżące saldo, a nie przez przypomnienie całej historii operacji na koncie.

W systemie złożonym z wielu obiektów rzadko się zdarza, żeby obiekty były całkowicie niezależne. Każdy z nich może wpływać na stany pozostałych obiektów poprzez interakcje służące do łączenia zmiennych stanu jednego obiektu ze zmiennymi stanu innych obiektów. W rzeczywistości pogląd, że system składa się z niezależnych obiektów, jest najbardziej przydatny wtedy, kiedy ze ściśle powiązanych zmiennych stanu systemu można utworzyć podsystemy, które są między sobą luźno powiązane.

Takie spojrzenie na system może być podstawą organizowania modeli obliczeniowych systemu. Aby tak powstały model był modularny, powinien być podzielony na obiekty obliczeniowe modelujące faktyczne obiekty w systemie. Każdy obiekt obliczeniowy musi mieć własne *lokalne zmienne stanu* opisujące faktyczny stan obiektu. Ponieważ stany obiektów w modelowanym systemie zmieniają się wraz z upływem czasu, zmienne stanu odpowiadających im obiektów obliczeniowych muszą również się zmieniać. Jeśli zdecydujemy się modelować upływ czasu w systemie poprzez upływ czasu w komputerze, to musimy dysponować sposobem tworzenia obiektów obliczeniowych, których zachowania zmieniają się w trakcie wykonywania programu. W szczególności, jeśli chcemy modelować zmienne stanu w języku programowania za pomocą zwykłych nazw symbolicznych, to język ten musi udostępniać *operator przypisania* (ang. *assignment operator*) pozwalający na zmianę wartości związanej z nazwą.

### 3.1.1. Lokalne zmienne stanu

Chcąc pokazać, co mamy na myśli, mówiąc o obiektach obliczeniowych charakteryzujących się stanami zmieniającymi się w czasie, zamodelujemy wypłacanie pieniędzy z konta bankowego. Użyjemy do tego procedury `withdraw`, której argumentem jest wypłacana kwota (`amount`). Jeśli na koncie jest wystarczająca ilość pieniędzy, aby móc dokonać wypłaty, to wynikiem `withdraw` powinno być saldo konta po dokonaniu wypłaty. W przeciwnym razie wynikiem `withdraw` powinien być komunikat *Brak środków na koncie*. Jeżeli na przykład na koncie jest początkowo 100 dolarów, to powinniśmy uzyskać następujące wyniki poniższych wywołań procedury `withdraw`:

(`withdraw 25`)

75

(`withdraw 25`)

50

(`withdraw 60`)

*"Brak środków na koncie"*

(`withdraw 15`)

35

Zauważmy, że dwukrotne obliczanie wartości wyrażenia (`withdraw 25`) daje za każdym razem inne wyniki. Jest to coś nowego, jeśli chodzi o zachowanie procedury. Jak dotąd wszystkie nasze procedury mogliśmy traktować jako opisy sposobów obliczania funkcji matematycznych. Wywołanie procedury powodowało obliczenie wartości funkcji dla podanych wartości argumentów,

a wyniki dwóch wywołań procedury dla takich samych wartości argumentów były zawsze takie same<sup>1</sup>.

Do zaimplementowania procedury `withdraw` możemy użyć zmiennej `balance` reprezentującej bieżące saldo konta i zdefiniować procedurę `withdraw` tak, aby korzystała z tej zmiennej. Procedura ta powinna sprawdzać, czy wartość `balance` jest przynajmniej tak duża jak kwota (`amount`), którą chcemy wypłacić. Jeśli tak, to `withdraw` powinna zmniejszyć `balance` o wypłacaną kwotę, a wynikiem jej wywołania powinna być nowa wartość `balance`. W przeciwnym razie wynikiem `withdraw` powinien być komunikat *Brak środków na koncie*. Oto definicje `balance` i `withdraw`:

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Brak środków na koncie"))
```

Zmniejszenie wartości `balance` dokonuje się na skutek obliczenia wyrażenia

```
(set! balance (- balance amount))
```

Korzystamy tu z formy specjalnej `set!` o następującej składni:

```
(set! <nazwa> <nowa wartość>)
```

gdzie `<nazwa>` jest symbolem, a `<nowa wartość>` jest dowolnym wyrażeniem. `Set!` zmienia znaczenie symbolu `<nazwa>` w taki sposób, że jego wartością jest wynik obliczenia wyrażenia `<nowa wartość>`. W powyższym przykładzie nowa wartość zmiennej `balance` powstaje przez odjęcie od jej starej wartości kwoty `amount`<sup>2</sup>.

Procedura `withdraw` używa również formy specjalnej `begin` do obliczenia wartości dwóch wyrażeń w przypadku, gdy warunek badany przez `if` jest speł-

<sup>1</sup> W rzeczywistości nie jest to do końca prawdą. Jednym z wyjątków był generator liczb losowych w punkcie 1.2.6. Pozostałe wyjątki dotyczyły zastosowania tablicy operacji i typów, wprowadzonej w punkcie 2.4.3, gdzie to, czy wyniki dwóch wywołań procedury `get` z takimi samymi wartościami argumentów były takie same, zależało od ewentualnych wywołań procedury `put` między jednym a drugim wywołaniem `get`. Jednakże dopóki nie wprowadzimy przypisania, nie jesteśmy w stanie sami utworzyć takich procedur.

<sup>2</sup> Wartość wyrażenia `set!` zależy od implementacji. Powinniśmy tak używać `set!`, aby polegać jedynie na efektach jego obliczania, a nie na jego wartości.

Nazwa `set!` jest zgodna z konwencją nazywania stosowaną w języku Scheme: nazwy operacji zmieniających wartości zmiennych (lub modyfikujących struktury danych, co zobaczymy w podrozdziale 3.3) mają na końcu wykryznik. Przypomina to konwencję nazywania predykatów, zgodnie z którą nazwy predykatów są zakończone znakiem zapytania.

niony: najpierw zmniejsza wartość zmiennej `balance`, a następnie przekazuje nową wartość tej zmiennej. Ogólnie mówiąc, obliczenie wyrażenia

`(begin ⟨e12k`

powoduje obliczenie kolejno wszystkich wyrażeń od `⟨e1 do ⟨ek, a wartość ostatniego z nich jest wartością całego wyrażenia begin3.`

Chociaż `withdraw` działa tak, jak chcieliśmy, zmienna `balance` sprawia pewne kłopoty. W powyższym programie `balance` jest zdefiniowana w środowisku globalnym i każda procedura może badać lub zmieniać jej wartość. Byłoby dużo lepiej, gdybyśmy mogli uczynić `balance` zmienną wewnętrzną względem `withdraw` — tak aby `withdraw` było jedyną procedurą mającą bezpośredni dostęp do `balance`, a wszystkie inne procedury miały jedynie pośredni dostęp do tej zmiennej (poprzez wywołania procedury `withdraw`). Lepiej oddawałoby to stwierdzenie, że `balance` jest lokalną zmienną stanu używaną przez `withdraw` do śledzenia stanu konta.

Możemy uczynić `balance` zmienną wewnętrzną względem `withdraw`, zapisując nasze definicje w następujący sposób:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Brak środków na koncie"))))
```

To, co tu zrobiliśmy, to użyliśmy `let` do utworzenia środowiska zawierającego zmienną lokalną `balance`, początkowo równą 100. Wewnątrz tego środowiska użyliśmy `lambda`-abstrakcji do utworzenia procedury, której argumentem jest wypłacana kwota (`amount`) i która zachowuje się tak jak poprzednia procedura `withdraw`. Procedura ta jest wynikiem obliczenia wyrażenia `let` i otrzymuje nazwę `new-withdraw`. Zachowuje się ona dokładnie tak samo jak `withdraw`, ale jej zmienna `balance` nie jest dostępna dla żadnej innej procedury<sup>4</sup>.

Stosowanie `set!` do zmiennych lokalnych stanowi ogólną technikę programowania, której będziemy używać do konstruowania obiektów obliczeniowych mających stany lokalne. Niestety, użycie tej techniki stwarza poważny problem.

<sup>3</sup> Używaliśmy już formy `begin` niejawnie w naszych programach, gdyż Scheme dopuszcza, aby treść procedury składała się z szeregu wyrażeń. Podobnie, następni klauzul w wyrażen夜里ach `cond` mogą być ciągami wyrażeń, a nie tylko pojedynczymi wyrażeniami.

<sup>4</sup> W żargonie informatycznym o takiej zmiennej mówi się, że jest *ukryta* (*zamknięta*; ang. *encapsulated*) *wewnętrz* procedury `new-withdraw`. Takie ukrywanie odzwierciedla bardziej ogólną zasadę projektowania systemów, znaną jako *zasada ukrywania informacji* (ang. *hiding principle*). Zgodnie z tą zasadą, jeżeli chcemy, aby nasze systemy były bardziej modularne i elastyczne, powinniśmy chronić różne części systemu przed sobą nawzajem, udostępniając jedynie te informacje, które „muszą być widoczne” na zewnątrz.

Wprowadzając na początku pojęcie procedury, wprowadziliśmy również podstawieniowy model obliczeń (punkt 1.1.5), określający znaczenie zastosowania procedury do argumentów. Powiedzieliśmy, że zastosowanie procedury należy interpretować jako obliczenie wartości treści procedury przy zastąpieniu parametrów formalnych ich wartościami. Problem polega na tym, że z chwilą, gdy wprowadzimy do naszego języka przypisanie, model podstawieniowy staje się niewystarczający do modelowania zastosowań procedur. (Dlaczego tak jest, dowiemy się w punkcie 3.1.3). W rezultacie, formalnie rzecz biorąc, nie dysponujemy w tej chwili żadnym narzędziem, które pozwoliłoby nam zrozumieć, dlaczego procedura `new-withdraw` zachowuje się tak, jak to opisaliśmy powyżej. Chcąc naprawdę zrozumieć działanie takich procedur jak `new-withdraw`, musimy najpierw opracować nowy model stosowania procedur. W podrozdziale 3.2 wprowadzimy taki model, a także wyjaśnimy realizację `set!` i zmiennych lokalnych. Przedtem jednak zbadamy kilka wariacji na temat procedury `new-withdraw`.

Następująca procedura `make-withdraw` tworzy „procesory wypłat”. Jej parametr formalny `balance` określa początkowe saldo konta<sup>5</sup>.

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie")))
```

Za pomocą procedury `make-withdraw` możemy w następujący sposób utworzyć dwa obiekty `W1` i `W2`:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))

(W1 50)
50

(W2 70)
30

(W2 40)
"Brak środków na koncie"

(W1 40)
10
```

---

<sup>5</sup> Przeciwnie niż w przypadku przedstawionej wcześniej procedury `new-withdraw` tutaj nie musimy używać `let` do utworzenia zmiennej lokalnej `balance`, gdyż parametry formalne są już lokalne. Stanie się to bardziej zrozumiałe po omówieniu w podrozdziale 3.2 środowiskowego modelu obliczeń. (Zobacz także ćwiczenie 3.10).

Zauważmy, że  $W_1$  i  $W_2$  są całkowicie niezależnymi obiektami i każdy z nich ma własną lokalną zmienną stanu `balance`. Operacje wypłaty wykonywane na jednym z nich nie zmieniają stanu drugiego.

Możemy także tworzyć obiekty udostępniające zarówno operacje wpłaty, jak i wypłaty — reprezentujące proste konta bankowe. Oto procedura, której wynikiem jest „obiekt konta bankowego” o określonym saldzie początkowym.

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Nieznaną operację -- MAKE-ACCOUNT"
                       m)))))

dispatch)
```

Każde wywołanie procedury `make-account` tworzy środowisko zawierające lokalną zmienną stanu `balance`. `make-account` definiuje wewnątrz tego środowiska procedury `deposit` i `withdraw`, modyfikujące zmienną `balance`, oraz dodatkową procedurę `dispatch`, której argumentem jest „komunikat”, a wynikiem jedna z dwóch procedur lokalnych. Procedura `dispatch` sama z kolei jest wynikiem reprezentującym konto bankowe jako obiekt obliczeniowy. Jest to dokładnie styl programowania oparty na *przekazywaniu komunikatów*, jaki widzieliśmy w punkcie 2.4.3, choć tutaj używamy go w połączeniu z możliwością modyfikowania zmiennych lokalnych.

Procedury `make-account` możemy używać w następujący sposób:

```
(define acc (make-account 100))

((acc 'withdraw) 50)
50

((acc 'withdraw) 60)
"Brak środków na koncie"

((acc 'deposit) 40)
90

((acc 'withdraw) 60)
30
```

Każde wywołanie `acc` daje w wyniku procedurę lokalną `deposit` lub `withdraw`, która jest następnie stosowana do określonej kwoty (`amount`). Tak jak w przypadku `make-withdraw`, kolejne wywołanie `make-account`:

```
(define acc2 (make-account 100))
```

utworzy całkowicie niezależny obiekt reprezentujący konto, zawierający własną zmienną lokalną `balance`.

### Ćwiczenie 3.1

*Akumulator* to jednoargumentowa procedura, którą wywołujemy wielokrotnie, przekazując jej kolejne wartości liczbowe. Procedura ta kumuluje te wartości, obliczając ich sumę. Za każdym razem, gdy ją wywołujemy, jej wynikiem jest aktualna suma. Napisz procedurę `make-accumulator` tworzącą akumulatory, z których każdy niezależnie oblicza swoją sumę. Argument `make-accumulator` powinien określać początkową wartość sumy; na przykład:

```
(define A (make-accumulator 5))
```

```
(A 10)
```

```
15
```

```
(A 10)
```

```
25
```

### Ćwiczenie 3.2

W systemach testujących oprogramowanie potrzebna jest czasem możliwość liczenia, ile razy w trakcie obliczeń dana procedura była wywoływana. Napisz procedurę `make-monitored`, której argumentem jest jednoargumentowa procedura `f`. Jej wynik jest również procedurą jednoargumentową (oznaczmy ją przez `mf`), która za pomocą wewnętrznego licznika zlicza, ile razy była wywoływana. Jeśli argument procedury `mf` jest symbolem specjalnym `how-many-calls?`, to jej wynikiem jest wartość licznika. Jeśli argument jest symbolem specjalnym `reset-count`, to `mf` zeruje swój licznik. W przypadku każdego innego argumentu wynikiem wywołania `mf` jest wynik wywołania `f` dla tego argumentu i zwiększenie licznika. Moglibyśmy na przykład utworzyć monitorowaną wersję procedury `sqrt`:

```
(define s (make-monitored sqrt))
```

```
(s 100)
```

```
10
```

```
(s 'how-many-calls?)
```

```
1
```

### Ćwiczenie 3.3

Zmodyfikuj procedurę `make-account` tak, aby tworzyła konta bankowe zabezpieczone hasłami. To znaczy, `make-account` powinna mieć dodatkowy argument; na przykład:

```
(define acc (make-account 100 'secret-password))
```

Tak powstały obiekt reprezentujący konto powinien wykonywać tylko takie operacje, którym towarzyszy takie samo hasło jak to podane w chwili utworzenia obiektu, a w przypadku innych haseł powinien sygnalizować błąd:

```
((acc 'secret-password 'withdraw) 40)
```

```
60
```

```
((acc 'some-other-password 'deposit) 50)
```

```
"Błędne hasło"
```

### Ćwiczenie 3.4

Zmodyfikuj procedurę `make-account` z ćwiczenia 3.3, dodając kolejną lokalną zmienią stanu, w taki sposób, że jeżeli siedem razy z rzędu zostanie podane błędne hasło, to nastąpi wywołanie procedury `call-the-cops` (dzwon-po-gliny).

#### 3.1.2. Korzyści z wprowadzenia przypisania

Jak zobaczymy, wprowadzenie przypisania do naszego języka programowania prowadzi do płatniny wielu trudnych pojęć. Niemniej jednak rozpatrywanie systemów jako kolekcji obiektów ze stanami lokalnymi jest potężną techniką pozwalającą na zachowanie modularności projektów. Jako prosty przykład rozważmy konstrukcję procedury `rand`, której wynikiem przy każdym wywołaniu jest losowa liczba całkowita.

Nie jest całkiem jasne, co to znaczy „losowa”. To, co zapewne chcielibyśmy osiągnąć, to procedura, której kolejne wywołania tworzą ciąg liczb o statystycznym rozkładzie jednostajnym. Nie będziemy się tutaj zajmować metodami generowania odpowiednich ciągów. Założymy raczej, że dostępna jest procedura `rand-update`, która na podstawie początkowej wartości  $x_1$  potrafi w następujący sposób utworzyć ciąg  $x_1, x_2, x_3, \dots$  o wymaganym rozkładzie statystycznym<sup>6</sup>:

<sup>6</sup> Jeden ze znanych sposobów zaimplementowania `rand-update` polega na zastosowaniu reguły, zgodnie z którą  $x$  jest przekształcane w  $ax + b$  modulo  $m$ , gdzie  $a, b$  i  $m$  są odpowiednio dobranymi liczbami całkowitymi. Rozdział 3 książki Knutha [59] zawiera szczegółowe omówienie technik generowania ciągów liczb losowych i badania ich własności statystycznych. Zauważmy, że procedura `rand-update` oblicza funkcję matematyczną — jeśli wywołamy ją dwa razy z tym samym argumentem, to otrzymamy takie same wyniki. Dlatego też ciąg liczb generowanych przez `rand-update` z pewnością nie jest „losowy”, jeżeli wymagamy, aby kolejna liczba w ciągu „losowym” nie zależała od liczb poprzedzających ją w tym ciągu. Różnica między „prawdziwą losością” i tzw. ciągami *pseudolosowymi*, tworzonymi przez dobrze określone

```
x2 = (rand-update x1)
x3 = (rand-update x2)
```

Możemy zaimplementować `rand` w postaci procedury z lokalną zmienną stanu `x`, inicjowaną pewną ustaloną wartością `random-init`. Każde wywołanie `rand` oblicza `rand-update` dla bieżącej wartości `x`, przekazuje wynik jako liczbę losową i zapamiętuje go jako nową wartość `x`.

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

Oczywiście moglibyśmy wygenerować taki sam ciąg liczb losowych bez użycia przypisania, po prostu wywołując bezpośrednio `rand-update`. To jednak oznaczałoby, że w każdej części naszego programu, w której używa się liczb losowych, musiałaby być jawnie pamiętaana i przekazywana do `rand-update` bieżąca wartość `x`. Aby zdać sobie sprawę, jak bardzo byłoby to irytujące, zastanówmy się nad użyciem liczb losowych do zaimplementowania techniki nazywanej *metodą Monte Carlo*.

Metoda Monte Carlo polega na losowaniu danych eksperymentalnych z dużego zbioru oraz wnioskowaniu na podstawie prawdopodobieństw oszacowanych za pomocą wyników obliczonych dla tych danych. Możemy na przykład przybliżać liczbę  $\pi$ , korzystając z faktu, że  $6/\pi^2$  jest prawdopodobieństwem, z jakim dwie losowe liczby całkowite są względnie pierwsze, tzn. ich największym wspólnym dzielnikiem jest 1<sup>7</sup>. W celu uzyskania przybliżonej wartości  $\pi$  wykonujemy wiele eksperymentów. Każdy z nich polega na wyborze dwóch losowych liczb całkowitych i sprawdzeniu, czy ich największy wspólny dzielnik jest równy 1. Proporcja liczby pozytywnych wyników testu do liczby wszystkich eksperymentów stanowi przybliżenie  $6/\pi^2$ , na podstawie którego uzyskujemy przybliżenie  $\pi$ .

Sercem naszego programu jest procedura `monte-carlo`, której argumentami są: liczba eksperymentów oraz eksperiment reprezentowany przez bezargumentową procedurę, której wynikiem za każdym razem jest prawda albo fałsz. Procedura `monte-carlo` wykonuje eksperiment określona liczbę razy, a jej wynikiem jest stosunek liczby eksperymentów, których wynikiem jest prawda, do liczby wszystkich eksperymentów.

---

ne obliczenia, a mimo to mającymi odpowiednie własności statystyczne, jest złożoną kwestią wiążącą się z trudnymi problemami matematycznymi i filozoficznymi. Kołmogorow, Solomonoff i Chaitin zrobili wiele na rzecz wyjaśnienia tych kwestii; rozważania na ten temat można znaleźć w [10].

<sup>7</sup> Twierdzenie to pochodzi od E. Cesàro. Omówienie i dowód tego twierdzenia można znaleźć w [59], punkt 4.5.2.

---

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test)))))

(define (cesaro-test)
  (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))
```

Spróbujmy teraz wykonać te same obliczenia, korzystając bezpośrednio z `rand-update` zamiast z `rand`, tak jak musielibyśmy to robić, gdybyśmy nie mogli użyć przypisania do modelowania stanów lokalnych:

```
(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init))))

(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0)
               (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1)
                     (+ trials-passed 1)
                     x2))
              (else
               (iter (- trials-remaining 1)
                     trials-passed
                     x2))))))
  (iter trials 0 initial-x))
```

Choć program ten pozostaje prosty, wykazuje pewne znamiona naruszenia modularności. W pierwszej wersji programu, z użyciem `rand`, mogliśmy wyrazić metodę Monte Carlo bezpośrednio jako ogólną procedurę `monte-carlo`, której argumentem mogła być dowolna procedura `experiment`. W drugiej wersji programu, nie zawierającej lokalnego stanu generatora liczb losowych, procedura `random-gcd-test` musi w jawnym sposób operować na liczbach losowych `x1` i `x2`, a także pamiętać wartość `x2` pochodząą z poprzedniego kroku pętli, aby móc ją przekazać do `rand-update`. Takie jawnie przekazy-

wanie liczb losowych powoduje, że w strukturę gromadzenia wyników testów został wpleciony fakt, iż w przypadku naszego eksperymentu używamy dwóch liczb losowych, podczas gdy inne eksperymenty Monte Carlo mogą wymagać tylko jednej liczby losowej lub trzech takich liczb. Nawet główna procedura `estimate-pi` musi zajmować się określeniem początkowej liczby losowej. To, że wewnętrzne szczegóły generatora liczb losowych przedostają się do innych części programu, utrudnia takie wydzielenie metody Monte Carlo, aby można ją było stosować do innych celów. W pierwszej wersji programu przypisanie ukrywa stan generatora liczb losowych wewnętrz procedury `rand` tak, że szczegóły generowania liczb losowych pozostają niezależne od reszty programu.

Na przykładzie metody Monte Carlo zilustrowaliśmy następujące ogólne zjawisko: Z punktu widzenia jednych części złożonego procesu pozostałe części zmieniają się wraz z upływem czasu. Mają one ukryte stany lokalne zmieniające się w czasie. Jeśli chcemy pisać programy komputerowe, których struktura odzwierciedla taki sposób dekompozycji, musimy tworzyć obiekty obliczeniowe (takie jak konta bankowe i generatory liczb losowych), których zachowanie zmienia się z upływem czasu. Stany tych obiektów modelujemy za pomocą lokalnych zmiennych stanu, a zmiany stanów realizujemy, przypisując tym zmiennym nowe wartości.

Kuszącą konkluzją tych rozważań mogłoby być stwierdzenie, że wprowadzenie przypisania i technika ukrywania stanów w zmiennych lokalnych pozwalają nadawać systemom bardziej modularną strukturę niż w przypadku, gdy trzeba w jawnym sposób operować wszystkimi stanami, przekazując je przez dodatkowe parametry. Niestety, jak zobaczymy, nie jest to takie proste.

### Ćwiczenie 3.5

*Całkowanie metodą Monte Carlo* polega na przybliżaniu całek oznaczonych za pomocą symulacji Monte Carlo. Rozważmy obliczenie powierzchni fragmentu płaszczyzny opisanego predykatem  $P(x, y)$ , który jest prawdziwy dla punktów  $(x, y)$  należących do tego fragmentu i fałszywy dla punktów spoza niego. Na przykład obszar w kształcie koła o promieniu 3 i środku w punkcie  $(5, 7)$  można opisać za pomocą predykatu  $(x - 5)^2 + (y - 7)^2 \leq 3^2$ . Chcąc oszacować powierzchnię obszaru opisanego predykatem, wybieramy najpierw prostokąt [o bokach równoległych do osi układu współrzędnych; przyp. tłum.] zawierający ten obszar. Na przykład prostokąt o wierzchołkach leżących po przekątnej w punktach  $(2, 4)$  i  $(8, 10)$  zawiera powyższe koło. Interesująca nas całka jest równa powierzchni tej części prostokąta, która należy do naszego obszaru. Możemy przybliżać wartość całki, wybierając wewnętrz prostokąta losowe punkty  $(x, y)$  i sprawdzając za pomocą predykatu  $P(x, y)$ , czy leżą one wewnętrz obszaru. Jeśli przeprowadzimy takie testy dla wielu punktów, to stosunek liczby punktów należących do obszaru do liczby wszystkich punktów da nam przybliżenie stosunku powierzchni obszaru do powierzchni prostokąta. Tak więc mnożąc uzyskany ułamek przez powierzchnię całego prostokąta, otrzymamy przybliżenie całki.

Zaimplementuj całkowanie metodą Monte Carlo w postaci procedury `estimate-integral`, której argumentami są: predykat  $P$ , współrzędne prostokąta  $x1, x2, y1$

i y2 oraz liczba testów, które należy wykonać w celu obliczenia przybliżenia. Twoja procedura powinna korzystać z tej samej procedury monte-carlo, która była zastosowana wcześniej do przybliżenia  $\pi$ . Użyj swojej procedury `estimate-integral` do przybliżenia  $\pi$  poprzez zmierzenie powierzchni koła jednostkowego.

Przyda Ci się procedura, której wynikiem jest liczba wybrana losowo z określonego przedziału. Następująca procedura `random-in-range` realizuje to za pomocą procedury `random`, użytej w punkcie 1.2.6, której wynikiem jest nieujemna liczba losowa mniejsza od podanego argumentu<sup>8</sup>.

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

### Ćwiczenie 3.6

Czasami przydatna jest możliwość ustawiania generatora liczb losowych tak, aby otrzymywać ciąg zaczynający się od zadanej wartości. Opracuj nową procedurę `rand` wywoływaną z argumentem, który jest albo symbolem `generate`, albo symbolem `reset` i działa w następujący sposób: wynikiem (`rand 'generate`) jest nowa liczba losowa; (`(rand 'reset) <nowa wartość>`) ustawia wewnętrzną zmienną stanu na podaną (`nową wartość`). Tak więc ustawiając stan, można generować powtarzalne ciągi. Jest to bardzo przydatne, zwłaszcza w trakcie testowania i usuwania błędów z programów używających liczb losowych.

#### 3.1.3. Koszty wprowadzenia przypisania

Jak widzieliśmy, operacja `set!` umożliwia modelowanie obiektów ze stanami lokalnymi. Jednakże ta korzyść ma swoją cenę. Nie możemy już dłużej interpretować naszego języka programowania za pomocą podstawieniowego modelu stosowania procedur, wprowadzonego w punkcie 1.1.5. Co więcej, żaden prosty model o „eleganckich” właściwościach matematycznych nie może stanowić podstawy zastosowania obiektów i przypisania w języku programowania.

Jak długo nie używamy przypisań, tak długo różne obliczenia tej samej procedury z tymi samymi argumentami będą dawać te same wyniki, dzięki czemu możemy patrzeć na procedury jak na funkcje matematyczne. Programowanie bez użycia przypisań, tak jak to robiliśmy w dwóch pierwszych rozdziałach niniejszej książki, jest nazywane *programowaniem funkcyjnym* (ang. *functional programming*).

Aby zrozumieć, jak duże komplikacje powoduje przypisanie, rozważmy uproszczoną wersję procedury `make-withdraw` z punktu 3.1.1, która nie sprawdza, czy na koncie są wystarczające środki:

---

<sup>8</sup> MIT Scheme udostępnia taką procedurę. Jeśli argumentem `random` jest liczba całkowita (jak w punkcie 1.2.6), to jej wynikiem jest również liczba całkowita, ale jeżeli otrzyma ona liczbę zmiennopozycyjną (jak w tym ćwiczeniu), to jej wynikiem jest również liczba zmiennopozycyjna.

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))

(define W (make-simplified-withdraw 25))

(W 20)
5

(W 10)
-5
```

Porównajmy tę procedurę z następującą procedurą `make-decrementer`, która nie korzysta z `set!`:

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

Wynikiem `make-decrementer` jest procedura, która odejmuje od określonej sumy `balance` podaną kwotę; jednakże rezultaty kolejnych wywołań nie kumulują się, jak w przypadku `make-simplified-withdraw`:

```
(define D (make-decrementer 25))

(D 20)
5

(D 10)
15
```

Działanie `make-decrementer` możemy wyjaśnić za pomocą modelu podstawienniowego. Przeanalizujmy na przykład obliczenie wyrażenia

```
((make-decrementer 25) 20)
```

Najpierw upraszczamy operator kombinacji, podstawiając w treści procedury `make-decrementer` 25 w miejsce `balance`. Upraszczamy to nasze wyrażenie do

```
((lambda (amount) (- 25 amount)) 20)
```

Teraz stosujemy operator, podstawiając w treści `lambda`-wyrażenia 20 w miejsce `amount`:

```
(- 25 20)
```

Ostatecznie otrzymujemy w wyniku 5.

Zauważmy jednak, co się stanie, jeśli spróbujemy zastosować podobną analizę podstawieniową do `make-simplified-withdraw`:

```
((make-simplified-withdraw 25) 20)
```

Najpierw upraszczamy operator kombinacji, podstawiając w treści procedury `make-simplified-withdraw` 25 w miejsce `balance`. Upraszczamy to nasze wyrażenie do<sup>9</sup>

```
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

Teraz stosujemy operator, podstawiając w treści `lambda`-wyrażenia 20 w miejsce `amount`:

```
(set! balance (- 25 20)) 25
```

Gdybyśmy stosowali się do modelu podstawieniowego, musielibyśmy stwierdzić, że zastosowanie procedury polega najpierw na przypisaniu `balance` wartości 5, a następnie na przekazaniu liczby 25 jako wyniku wyrażenia. Prowadzi to jednak do złego wyniku. W celu uzyskania poprawnego wyniku, musielibyśmy jakoś odróżnić pierwsze wystąpienie `balance` (przed zastosowaniem `set!`) od drugiego (po zastosowaniu `set!`), czego nie możemy zrobić w modelu podstawieniowym.

Problem polega tu na tym, że podstawienie bezwzględnie korzysta z tego, że symbole w naszym języku programowania są w gruncie rzeczy nazwami wartości. Ale gdy tylko wprowadzimy `set!` i pozwolimy, aby wartość zmiennej mogła się zmieniać, wówczas zmienna nie może dłużej być po prostu nazwą. Od tej pory zmienna w jakiś sposób odwołuje się do miejsca, w którym wartość może być przechowywana, a wartość przechowywana w tym miejscu może się zmieniać. W podrozdziale 3.2 zobaczymy, w jaki sposób środowiska odgrywają rolę takich „miejsc” w modelu obliczeń.

### Niezmienność i zmiana

Kwestią, którą tutaj poruszamy, jest gębsza niż zwykle załamanie się danego modelu obliczeń. Z chwilą, gdy do naszych modeli obliczeń wprowadzamy możliwość zmian, wiele pojęć, które były dotychczas oczywiste, staje się problematycznych. Rozważmy stwierdzenie, że dwie rzeczy są „takie same”.

Przypuśćmy, że wywołujemy dwukrotnie `make-decrementer` z takim samym argumentem, tworząc dwie procedury:

```
(define D1 (make-decrementer 25))
```

```
(define D2 (make-decrementer 25))
```

---

<sup>9</sup> Nie zastępujemy wystąpienia `balance` w wyrażeniu `set!`, gdyż `(nazwa)` w `set!` nie jest obliczana. Gdybyśmy dokonali takiego podstawienia, otrzymalibyśmy `(set! 25 (- 25 amount))`, co nie ma sensu.

Czy D1 i D2 są takie same? Dopuszczalna jest odpowiedź twierdząca, gdyż D1 i D2 wykazują takie samo zachowanie obliczeniowe — każda z nich jest procedurą odejmującą swój argument od 25. W rzeczywistości moglibyśmy w dowolnym obliczeniu zastąpić D1 przez D2 i nie zmieniłoby to wyników.

Przeciwstawmy to dwóm wywołaniom `make-simplified-withdraw`:

```
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-withdraw 25))
```

Czy W1 i W2 są takie same? Z pewnością nie, gdyż wywołania W1 i W2 dają różne rezultaty, jak pokazuje to następujący ciąg interakcji:

```
(W1 20)
5
```

```
(W1 20)
-15
```

```
(W2 20)
5
```

Chociaż W1 i W2 są „równe” w takim sensie, że powstały w wyniku obliczenia tego samego wyrażenia, (`make-simplified-withdraw 25`), nie jest prawdą, że w dowolnym wyrażeniu moglibyśmy zastąpić W1 przez W2 i nie zmieniłoby to wyników obliczenia tego wyrażenia.

Jeżeli w języku możemy w wyrażeniach „zastępować jedne elementy innymi, równymi im elementami”, nie zmieniając przy tym wartości wyrażeń, to mówimy, że taki język jest *niewrażliwy na odniesienia* (ang. *referentially transparent*). Niewrażliwość na odniesienia zostaje naruszona w momencie wprowadzenia `set!` do języka programowania. Na skutek tego nie jest oczywiste, kiedy możemy upraszczać wyrażenia, podstawiając równoważne im wyrażenia. W rezultacie wnioskowanie o programach używających przypisania staje się dużo bardziej skomplikowane.

Gdy już zrezygnujemy z niewrażliwości na odniesienia, trudno jest formalnie zdefiniować, co to znaczy, że obiekty obliczeniowe są „takie same”. Prawdę powiedziałbym, znaczenie „bycia takim samym” w rzeczywistym świecie, modelowanym przez nasze programy, wcale nie jest jasne. Mówiąc ogólnie, możemy stwierdzić, czy dwa najwyraźniej identyczne obiekty są faktycznie „takie same”, jedynie modyfikując jeden z nich i sprawdzając, czy drugi obiekt zmienił się tak samo. Jak jednak możemy sprawdzić, czy obiekt „się zmienił”, inaczej niż obserwując „ten sam” obiekt dwukrotnie i sprawdzając, czy jakaś jego własność nie uległa zmianie? Nie możemy więc stwierdzić „zmiany” bez wprowadzenia a priori pojęcia „tożsamości” (niezmienności) i nie możemy stwierdzić tożsamości bez obserwowania zmian.

Jako przykład występowania tego problemu w programowaniu rozważmy sytuację, gdy Piotr i Paweł mają konto bankowe, a na nim 100 dolarów. Jest zasadnicza różnica między zamodelowaniem tej sytuacji jako

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

a zamodelowaniem jej jako

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

W pierwszej sytuacji są to dwa różne konta bankowe. Transakcje dokonywane przez Piotra nie wpływają na stan konta Pawła i odwrotnie. W drugiej sytuacji jednak zdefiniowaliśmy paul-acc jako *to samo* co peter-acc. W rezultacie Piotr i Paweł mają teraz wspólne konto bankowe i jeśli Piotr dokona wypłaty z peter-acc, to Paweł stwierdzi, że ma mniej pieniędzy na koncie paul-acc. Te dwie sytuacje — podobne, ale jednak różne — mogą powodować zamęt przy budowaniu modeli obliczeniowych. W szczególności, w przypadku wspólnego konta może być wyjątkowo mylące, że mamy jeden obiekt (konto bankowe) występujący pod dwiema nazwami (peter-acc i paul-acc); szukając w programie wszystkich miejsc, w których paul-acc może ulec zmianie, musimy również znaleźć wszystkie te miejsca, w których peter-acc może ulec zmianie<sup>10</sup>.

Pamiętając o powyższych uwagach dotyczących „tożsamości” i „zmiany”, zauważmy, że gdyby Piotr i Paweł mogli tylko sprawdzać salda swoich kont, a nie mogliby wykonywać operacji zmieniających saldo, wówczas kwestia, czy ich konta są odrębne, byłaby sporna. Ogólnie mówiąc, dopóki nie modyfikujemy obiektów danych, dopóty możemy uważać, że obiekty złożone są dokładnie sumami swoich składowych. Liczba wymierna na przykład jest określona przez jej licznik i mianownik. Ale taki punkt widzenia traci rację bytu z chwilą dopuszczenia zmian, gdy złożony obiekt danych ma „tożsamość” będącą czymś innym niż składowe, z których jest złożony. Konto bankowe pozostaje „tym samym” kontem, nawet jeśli zmienimy jego saldo, dokonując wypłaty; i odwrotnie, możemy mieć dwa różne konta bankowe zawierające te same infor-

<sup>10</sup> Zjawisko polegające na tym, że jeden obiekt obliczeniowy może występować pod więcej niż jedną nazwą jest znane jako *utożsamianie nazw* (*aliasowanie*; ang. *aliasing*). Wspólne konto bankowe stanowi bardzo prosty przykład utożsamiania nazw. W podrozdziale 3.3 zobaczymy dużo bardziej skomplikowane przykłady, takie jak „różne” złożone struktury danych, które współdzielą swoje części. W naszych programach mogą pojawić się błędy, jeśli zapomnimy o tym, że zmiana jednego obiektu może również powodować jako „efekt uboczny” zmianę „innego” obiektu, w sytuacji gdy dwa „różne” obiekty są faktycznie jednym obiektem występującym pod dwiema różnymi nazwami. Te tzw. *błędy spowodowane efektami ubocznymi* są tak trudne do znalezienia i zanalizowania, że zaproponowano, aby języki programowania były tworzone w taki sposób, żeby nie dopuszczały ani efektów ubocznych, ani utożsamiania nazw [65, 80].

macje o ich stanie. Szkopuł ten nie wynika z naszego języka programowania, ale z naszego sposobu postrzegania konta bankowego jako obiektu. Nie traktujemy na przykład liczby wymiernej jako obiektu posiadającego tożsamość i mogącego ulegać zmianom w taki sposób, że moglibyśmy zmienić licznik i nadal mieć „tę samą” liczbę wymierną.

### Pułapki programowania imperatywnego

W przeciwieństwie do programowania funkcyjnego programowanie korzystające w znacznym stopniu z przypisań jest znane jako *programowanie imperatywne* (ang. *imperative programming*). Pomijając problemy związane z modelami obliczeń, programy pisane w stylu imperatywnym są podatne na błędy, które nie występują w programach funkcyjnych. Przypomnijmy sobie na przykład z punktu 1.2.1 iteracyjny program obliczający silnię:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Zamiast przekazywać argumenty w wewnętrznej pętli iteracyjnej, moglibyśmy zastosować bardziej imperatywny styl i użyć jawnego przypisania w celu aktualnienia wartości zmiennych `product` i `counter`:

```
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

Nie zmienia to wyników programu, ale wprowadza subtelną pułapkę. W jakiej kolejności są wykonywane przypisania? Tak się akurat składa, że program jest poprawny. Jednak odwrócenie kolejności przypisań:

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

dawałoby inne, niepoprawne wyniki. Ogólnie mówiąc, programowanie z użyciem przypisań zmusza nas do uważnego rozważenia wzajemnych kolejności

przypisań, aby się upewnić, że każde przypisanie korzysta z właściwych wartości zmiennych podlegających zmianom. Problem ten po prostu nie pojawia się w programach funkcyjnych<sup>11</sup>.

Złożoność programów imperatywnych staje się jeszcze większa, jeśli rozważamy zastosowania, w których wiele procesów jest wykonywanych równolegle. Wróćmy do tego w podrozdziale 3.4. Najpierw jednak zajmiemy się opracowaniem modelu obliczeniowego dla wyrażeń zawierających przypisania i zbadamy zastosowania obiektów ze stanami lokalnymi przy konstruowaniu symulacji.

### Ćwiczenie 3.7

Rozważmy utworzone przez `make-account` obiekty konta bankowego zabezpieczone hasłami, opisane w ćwiczeniu 3.3. Przypuśćmy, że w naszym systemie bankowym wymagana jest możliwość tworzenia kont wspólnych. Zdefiniuj realizującą to procedurę `make-joint`. Powinna ona mieć trzy argumenty. Pierwszym z nich jest zabezpieczone hasłem konto. Drugi argument musi być hasłem określonym w momencie utworzenia konta, w przeciwnym razie operacja `make-joint` powinna przerwać działanie. Trzeci argument jest nowym hasłem. `Make-joint` powinna utworzyć dodatkowy dostęp do konta, zabezpieczony nowym hasłem. Jeśli na przykład `peter-acc` jest kontem bankowym zabezpieczonym hasłem `open-sesame`, to

```
(define paul-acc
  (make-joint peter-acc 'open-sesame 'rosebud))
```

pozwoli na wykonywanie operacji na `peter-acc` za pomocą nazwy `paul-acc` i hasła `rosebud`. Możesz chcieć zmodyfikować rozwiązanie ćwiczenia 3.3, dodając do niego tę nową funkcję.

### Ćwiczenie 3.8

Gdy w punkcie 1.1.3 zdefiniowaliśmy model obliczeń, powiedzieliśmy, że pierwszy krok obliczania wyrażenia polega na obliczeniu jego podwyrażeń. Nie powiedzieliśmy jednak nigdzie, w jakiej kolejności podwyrażenia powinny być obliczane (np. od lewej do prawej czy odwrotnie). Kiedy wprowadzamy przypisanie, kolejność obliczania argumentów procedury może mieć wpływ na wynik. Zdefiniuj prostą procedurę `f`, taką że obliczenie `(+ (f 0) (f 1))` da w wyniku: 0, jeśli argumenty `+` są obliczane od lewej do prawej, lub 1, jeśli są obliczane od prawej do lewej.

<sup>11</sup> Biorąc to pod uwagę, zakrawa na ironię fakt, że często uczy się wstępu do programowania, używając wybitnie imperatywnego stylu. Może to być skutkiem pozostałości opinii panujących w latach sześćdziesiątych i siedemdziesiątych XX w., że programy wywołujące procedury muszą być z natury mniej efektywne niż programy wykonujące przypisania. (Steele [92] obala ten argument). Może to również odzwierciedlać pogląd, że początkującym programistom łatwiej jest wyobrazić sobie wykonywane krok po kroku przypisania niż wywołanie procedury. Bez względu na przyczynę obarcza to zwykle początkujących programistów problemami typu „czy powiniensem najpierw przypisać wartość tej, czy innej zmiennej”, co komplikuje tylko programowanie i zaciemnia najważniejsze koncepcje.

### 3.2. Środowiskowy model obliczeń

Gdy w rozdziale 1 wprowadziliśmy procedury złożone, użyliśmy podstawionego modelu obliczeń (punkt 1.1.5) w celu zdefiniowania znaczenia zastosowania procedury do argumentów:

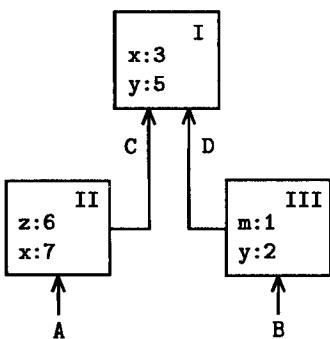
- Aby zastosować procedurę złożoną do argumentów, należy obliczyć wartość treści procedury z parametrami formalnymi zastąpionymi przez odpowiadające im argumenty.

Z chwilą dopuszczenia możliwości używania w naszym języku programowania przypisań taka definicja przestaje być wystarczająca. W szczególności, w punkcie 3.1.3 dowiedliśmy, że w obecności przypisania zmienna nie może już być traktowana jako zwykła nazwa wartości. Zamiast tego zmienna musi jakoś wyznaczać „miejsce”, w którym mogą być przechowywane wartości. W naszym nowym modelu obliczeń miejsca te będziemy trzymać w strukturach nazywanych *środowiskami* (ang. *environments*).

Środowisko to ciąg *namek* (ang. *frames*). Każda ramka to tablica (być może pusta) zawierająca *wiązania* (ang. *bindings*) przyporządkowujące nazwom zmiennych odpowiadające im wartości. (Jedna ramka może zawierać co najwyżej jedno wiązanie dla każdej zmiennej). Ponadto każda ramka zawiera wskaźnik do jej *środowiska otaczającego* (ang. *enclosing environment*), z wyjątkiem (na potrzeby niniejszej dyskusji) ramki *globalnej*. Wartość zmiennej w danym środowisku jest określona przez wiązanie danej zmiennej, znajdującej się w pierwszej ramce środowiska zawierającej takie wiązanie. Jeśli żadna ramka w ciągu nie zawiera wiązania zmiennej, to mówimy, że zmienna jest *wolna* (ang. *free*) w środowisku.

Na rysunku 3.1 widać prostą strukturę środowisk złożoną z trzech ramek oznaczonych I, II i III. Litery A, B, C i D reprezentują wskaźniki do środowisk. C i D wskazują na to samo środowisko. W ramce II znajdują się wiązania zmiennych z i x, a w ramce I — wiązania zmiennych y i x. Wartością x w środowisku D jest 3. Wartością x w odniesieniu do środowiska B również jest 3. Określa się to w następujący sposób. Badamy pierwszą ramkę w ciągu (ramka III) i nie znajdujemy tam wiązania x, więc przechodzimy do otaczającego środowiska D, gdzie w ramce I znajduje się szukane wiązanie. Z kolei wartość x w środowisku A jest równa 7, gdyż pierwsza ramka w ciągu (ramka II) zawiera wiązanie przyporządkowujące x wartość 7. Mówimy, że w odniesieniu do środowiska A wiązanie łączące zmienną x z wartością 7 w ramce II przesłania wiązanie znajdujące się w ramce I, przyporządkowujące x wartość 3.

Pojęcie środowiska ma zasadnicze znaczenie dla procesu obliczania, gdyż określa ono kontekst, w jakim wyrażenie powinno być obliczane. Właściwie



Rys. 3.1. Prosta struktura środowisk

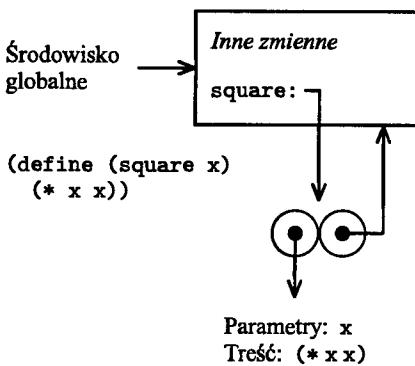
można powiedzieć, że wyrażenia w języku programowania same w sobie nie mają żadnego sensu. Nabierają dopiero znaczenia w kontekście środowiska, w którym są obliczane. Nawet interpretacja tak prostych wyrażeń jak  $(+ 1 1)$  opiera się na założeniu, że w kontekście, w którym operujemy,  $+$  jest symbolem oznaczającym dodawanie. Tak więc w naszym modelu obliczeń będziemy zawsze mówić o obliczaniu wyrażenia w odniesieniu do pewnego środowiska. Aby opisać interakcje z interpreterem, będziemy zakładać, że istnieje środowisko globalne, składające się z jednej ramki (bez środowiska otaczającego) zawierającej wartości dla symboli związanych z procedurami pierwotnymi. Na przykład fakt, że  $+$  jest symbolem oznaczającym dodawanie, można ująć, mówiąc, że w środowisku globalnym symbol  $+$  jest związany z pierwotną procedurą dodawania.

### 3.2.1. Reguły obliczania

Ogólna specyfikacja sposobu obliczania kombinacji przez interpreter pozostaje taka sama jak ta przedstawiona po raz pierwszy w punkcie 1.1.3:

- Aby obliczyć wartość kombinacji, należy wykonać następujące czynności:
  1. Obliczyć wartości podwyrażeń kombinacji<sup>12</sup>.
  2. Zastosować procedurę będącą wartością skrajnie lewego podwyrażenia (operatora) do argumentów będących wartościami pozostałych podwyrażeń.

<sup>12</sup> Przypisanie wprowadza pewne subtelności do pierwszego kroku reguły obliczania. Jak było pokazane w ćwiczeniu 3.8, obecność przypisania pozwala na napisanie wyrażeń, których wartość zależy od kolejności obliczania podwyrażeń kombinacji. Tak więc, aby być dokładnym, trzeba w pierwszym kroku określić kolejność obliczania (np. od lewej do prawej lub od prawej do lewej). Jednakże kolejność ta powinna być zawsze traktowana jako szczegół implementacyjny i nigdy nie należy pisać programów zależnych od jakiejś określonej kolejności. Na przykład wyrafinowany kompilator może optymalizować program, zmieniając kolejność, w której obliczane są podwyrażenia.



Rys. 3.2. Struktura środowisk powstała w wyniku obliczenia `(define (square x) (* x x))` w środowisku globalnym

Środowiskowy model obliczeń zastępuje model podstawieniowy, określając znaczenie zastosowania procedury złożonej do argumentów.

W środowiskowym modelu obliczeń procedura jest zawsze parą składającą się z kodu i wskaźnika do środowiska. Procedury są tworzone tylko w jeden sposób: przez obliczanie lambda-wyrażeń. Powstaje wówczas procedura, której kod pochodzi z treści lambda-wyrażenia i której środowisko jest środowiskiem, w którym obliczane było lambda-wyrażenie tworzące procedurę. Rozważmy na przykład definicję procedury

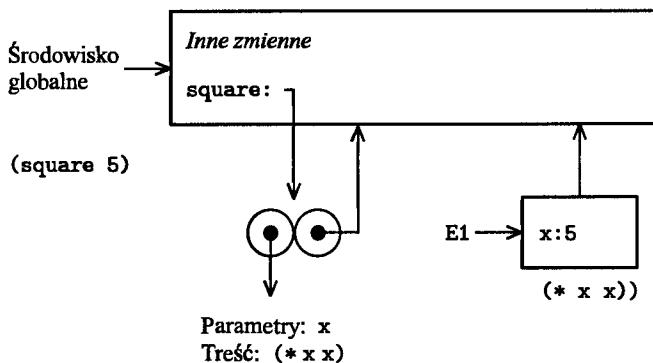
```
(define (square x)
  (* x x))
```

obliczaną w środowisku globalnym. Składnia definicji procedury jest tylko lukrem syntaktycznym pokrywającym niejawne lambda-wyrażenie. Jest ona równoważna definicji

```
(define square
  (lambda (x) (* x x)))
```

która oblicza `(lambda (x) (* x x))` i wiąże `square` z wynikiem — wszystko w środowisku globalnym.

Na rysunku 3.2 jest pokazany wynik obliczenia tego wyrażenia `define`. Obiekt proceduralny jest parą, która zawiera kod określający, że procedura ma jeden parametr formalny, mianowicie `x`, i treść `(* x x)`. Zawiera ona również wskaźnik do środowiska globalnego, gdyż to właśnie w tym środowisku obliczane było lambda-wyrażenie, którego wynikiem jest procedura. Nowe wiązanie, przyporządkowujące obiekt proceduralny symbolowi `square`, zostało dodane do ramki globalnej. Ogólnie mówiąc, `define` tworzy definicje obiektów, dodając wiązania do ramek.



Rys. 3.3. Środowisko powstałe w wyniku obliczenia wyrażenia (square 5) w środowisku globalnym

Teraz, gdy już zobaczyliśmy, jak tworzy się procedury, możemy opisać, jak się je stosuje. Model środowiskowy stanowi: Aby zastosować procedurę do argumentów, należy utworzyć nowe środowisko zawierające ramkę, która wiąże parametry z wartościami argumentów. Środowiskiem otaczającym dla tej ramki powinno być środowisko określone przez procedurę. Następnie w tym nowym środowisku trzeba obliczyć treść procedury.

Działanie tej reguły możemy prześledzić na rys. 3.3, gdzie jest pokazana struktura środowisk powstała w wyniku obliczenia wyrażenia (square 5) w środowisku globalnym, przy czym square jest procedurą przedstawioną na rys. 3.2. Zastosowanie procedury powoduje powstanie nowego środowiska, oznaczonego na rysunku przez E1, którego pierwsza ramka wiąże x (parametr formalny procedury) z argumentem 5. Wskaźnik prowadzący od tej ramki do góry wskazuje, że środowiskiem otaczającym ramki jest środowisko globalne. Jest to akurat środowisko globalne, gdyż właśnie to środowisko jest częścią obiektu proceduralnego square. Treść procedury, (\* x x), obliczamy w środowisku E1. Ponieważ wartość x w E1 wynosi 5, wynik jest równy (\* 5 5), czyli 25.

Środowiskowy model stosowania procedur można podsumować dwiema regułami:

- Zastosowanie obiektu proceduralnego do argumentów polega na utworzeniu ramki, powiązaniu formalnych parametrów procedury z argumentami wywołania i obliczeniu treści procedury w kontekście tak skonstruowanego nowego środowiska. Środowiskiem otaczającym nową ramkę jest środowisko będące składową stosowanego obiektu proceduralnego.
- Procedura jest tworzona przez obliczanie w danym środowisku lambda-wyrażenia. Powstający obiekt proceduralny jest parą składającą się z treści lambda-wyrażenia i wskaźnika do środowiska, w którym procedura została utworzona.

Precyzujemy także, że zdefiniowanie symbolu za pomocą `define` powoduje utworzenie w bieżącej ramce środowiska wiązania i przypisanie temu symbolowi wskazanej wartości<sup>13</sup>. Na koniec, określamy działanie `set!` — operacji, która w ogóle zmusiła nas do wprowadzenia modelu środowiskowego. Obrócenie w danym środowisku wyrażenia postaci (`set! <zmienna> <wartość>`) powoduje znalezienie wiązania zmiennej w środowisku i zmianę tego wiązania, aby wskazywało nową wartość. To znaczy, znajdujemy pierwszą ramkę w środowisku, która zawiera wiązanie dla tej zmiennej, i modyfikujemy tę ramkę. Jeśli zmienna nie jest związana w środowisku, to `set!` zgłasza błąd.

Te reguły obliczania, mimo że znacznie bardziej skomplikowane niż w modelu podstawieniowym, są nadal w miarę proste. Co więcej, taki model obliczeń, choć abstrakcyjny, stanowi poprawny opis sposobu obliczania wyrażeń przez interpreter. W rozdziale 4 zobaczymy, jak model ten może posłużyć za projekt działającego interpretera. W kolejnych punktach zajmujemy się szczegółami modelu obliczeń, analizując kilka przykładowych programów.

### 3.2.2. Stosowanie prostych procedur

Gdy w punkcie 1.1.5 wprowadziliśmy model podstawieniowy, pokazaliśmy, jak w wyniku obliczenia kombinacji (`f 5`) otrzymujemy liczbę 136 przy następujących definicjach procedur:

```
(define (square x)
  (* x x))

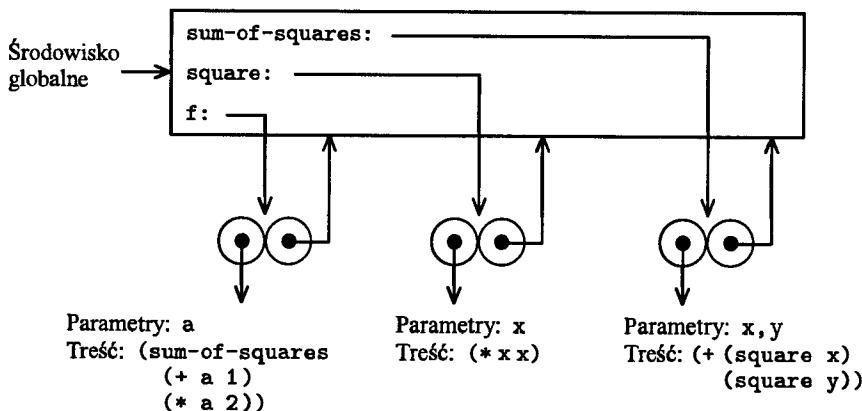
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

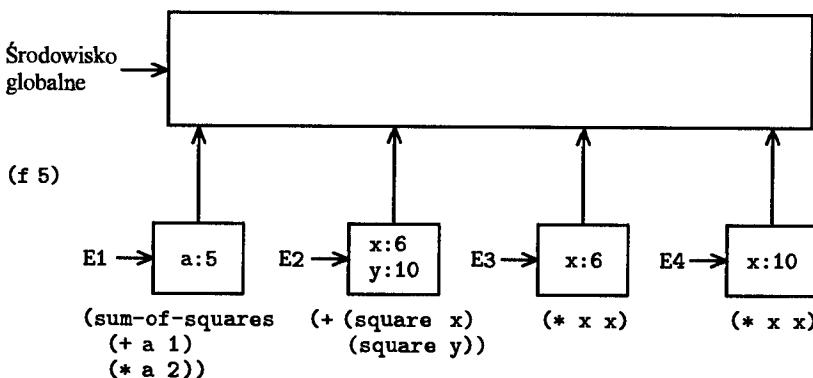
Możemy zanalizować ten sam przykład, stosując model środowiskowy. Na rysunku 3.4 są pokazane trzy obiekty proceduralne utworzone przez obliczenie definicji `f`, `square` i `sum-of-squares` w środowisku globalnym. Każdy obiekt proceduralny składa się z kodu oraz wskaźnika do środowiska globalnego.

Na rysunku 3.5 widać strukturę środowisk utworzonych przez obliczenie wyrażenia (`f 5`). Wywołanie procedury `f` tworzy nowe środowisko E1, któ-

<sup>13</sup> Jeśli w bieżącej ramce istnieje już wiązanie dla zmiennej, to ulega ono zmianie. Jest to dogodny mechanizm, gdyż pozwala na przedefiniowywanie symboli; oznacza to jednak również, że można użyć `define` do zmiany wartości, co doprowadza nas do kwestii przypisania bez jawnego używania `set!`. Dlatego też niektórzy wolą, aby przedefiniowanie istniejącego symbolu powodowało błąd lub generowało ostrzeżenie.



Rys. 3.4. Obiekty proceduralne znajdujące się w ramce globalnej



Rys. 3.5. Środowiska utworzone przez obliczenie wyrażenia (f 5) przy wykorzystaniu procedur z rys. 3.4

rego pierwsza ramka wiąże parametr formalny a tej procedury z argumentem 5. W środowisku E1 obliczamy treść procedury f:

(sum-of-squares (+ a 1) (\* a 2))

Chcąc obliczyć tę kombinację, najpierw obliczamy wartości podwyrażeń. Wartość pierwszego podwyrażenia, sum-of-squares, jest obiektem proceduralnym. (Zwróćmy uwagę, jak ta wartość jest znajdowana: najpierw sprawdzamy pierwszą ramkę środowiska E1, w której jednak nie ma wiązania dla sum-of-squares, a następnie sprawdzamy środowisko otaczające, w tym przypadku środowisko globalne, gdzie znajdujemy wiązanie pokazane na rys. 3.4). Pózostałe dwa podwyrażenia obliczamy, używając operacji pierwotnych + i \* do obliczenia dwóch kombinacji (+ a 1) i (\* a 2), których wynikami są odpowiednio 6 i 10.

Teraz stosujemy obiekt proceduralny `sum-of-squares` do argumentów 6 i 10. W rezultacie powstaje nowe środowisko E2, w którym parametry formalne `x` i `y` są związane z argumentami. W środowisku E2 obliczamy kombinację `(+ (square x) (square y))`. Prowadzi to do obliczenia `(square x)`, przy czym `square` jest określone w ramce globalnej, a `x` jest równe 6. Po raz kolejny tworzymy nowe środowisko, E3, w którym `x` jest związane z liczbą 6, i w tym środowisku obliczamy treść `square`, czyli `(* x x)`. W ramach zastosowania `sum-of-squares` musimy również obliczyć podwyrażenie `(square y)`, dla `y` równego 10. To drugie wywołanie `square` tworzy kolejne środowisko, E4, w którym parametr formalny `x` procedury `square` jest równy 10. W środowisku tym musimy obliczyć `(* x x)`.

Ważną kwestią, na którą należy zwrócić uwagę, jest to, że każde wywołanie `square` tworzy nowe środowisko zawierające wiązanie dla `x`. Widać tutaj, jak różne ramki umożliwiają oddzielenie od siebie różnych zmiennych lokalnych — wszystkich o nazwie `x`. Zauważmy, że każda z ramek utworzonych przez `square` wskazuje na środowisko globalne, gdyż jest to środowisko, na które wskazuje obiekt proceduralny `square`.

Po obliczeniu podwyrażeń przekazywane są wyniki. Wartości wygenerowane przez oba wywołania `square` są dodawane przez `sum-of-squares`, a wynik tego dodawania jest również wynikiem procedury `f`. Ponieważ koncentrujemy się tutaj na strukturze środowisk, nie będziemy zajmować się tym, w jaki sposób wyniki są przekazywane między wywołaniami — jest to jednak ważny element procesu obliczania i zajmiemy się nim dokładniej w rozdziale 5.

### Ćwiczenie 3.9

W punkcie 1.2.1 używaliśmy modelu podstawieniowego do analizy dwóch procedur obliczających silnię: rekurencyjnej

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

oraz iteracyjnej

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Pokaż, jaka jest struktura środowisk powstających podczas obliczania (`factorial 6`) za pomocą obydwu wersji procedury `factorial`<sup>14</sup>.

### 3.2.3. Ramki jako magazyny stanów lokalnych

Możemy teraz się przyjrzeć, jak w modelu środowiskowym można użyć procedur i przypisania do reprezentowania obiektów ze stanem lokalnym. Jako przykład rozważmy „procesor wypłat”, przedstawiony w punkcie 3.1.1, tworzonej przez procedurę

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie")))
```

Opiszmy obliczenie

```
(define W1 (make-withdraw 100))
```

po którym następuje

```
(W1 50)
50
```

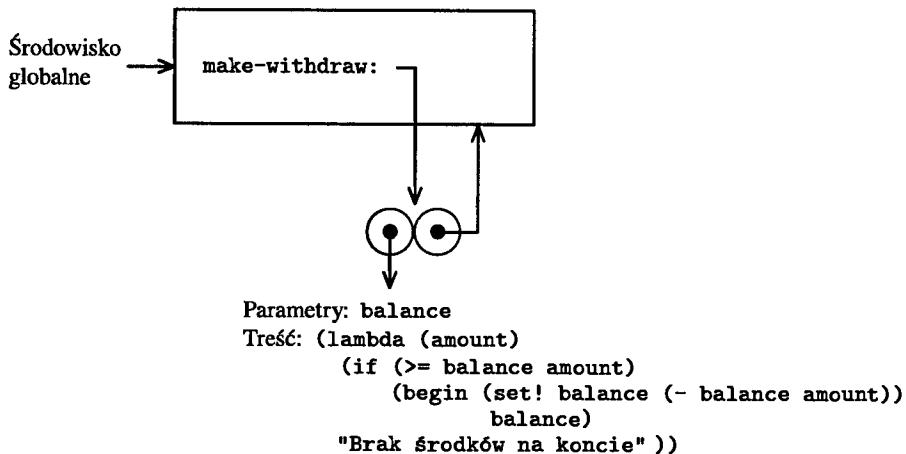
Na rysunku 3.6 jest przedstawiony wynik zdefiniowania procedury `make-withdraw` w środowisku globalnym. Jest to obiekt proceduralny zawierający wskaźnik do środowiska globalnego. Jak dotąd nie różni się on niczym od dotychczasowych przykładów z wyjątkiem tego, że treść procedury sama jest `lambda-wyrażeniem`.

Najciekawsza część obliczenia następuje, gdy zastosujemy procedurę `make-withdraw` do argumentu:

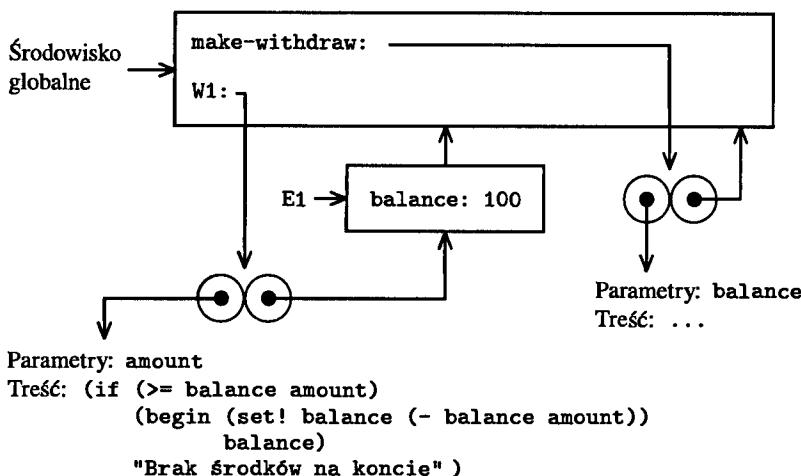
```
(define W1 (make-withdraw 100))
```

Jak zwykle zaczynamy od utworzenia środowiska `E1`, w którym parametr formalny `balance` jest związany z argumentem `100`. W tym środowisku obliczamy treść `make-withdraw`, a mianowicie `lambda-wyrażenie`. W rezultacie powstaje nowy obiekt proceduralny, którego kod określa `lambda` i którego środowiskiem jest `E1` — środowisko, w którym zostało obliczone `lambda-wyrażenie` tworzące procedurę. Tak utworzony obiekt obliczeniowy jest wynikiem

<sup>14</sup> Model środowiskowy nie wyjaśnia naszego stwierdzenia z punktu 1.2.1, że interpreter może wykonać taką procedurę jak `fact-iter` w stałej ilości pamięci, stosując rekursję ogonową. Rekursję ogonową omówimy w podrozdziale 5.4, kiedy będziemy się zajmować strukturą sterującą interpretera.



Rys. 3.6. Wynik zdefiniowania `make-withdraw` w środowisku globalnym



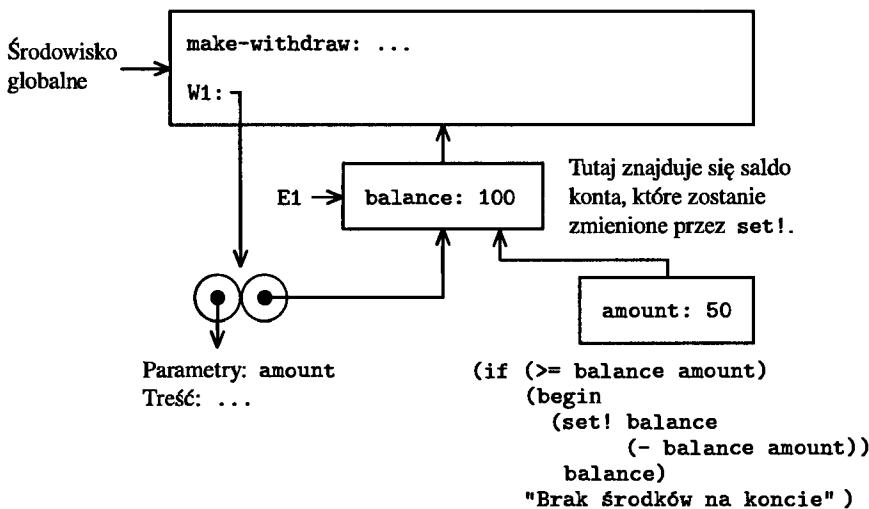
Rys. 3.7. Wynik obliczenia `(define W1 (make-withdraw 100))`

`make-withdraw`. Jest on powiązany w środowisku globalnym z `W1`, gdyż samo `define` jest obliczane w środowisku globalnym. Na rysunku 3.7 widać uzyskaną strukturę środowisk.

Możemy teraz przeanalizować, co się dzieje, gdy `W1` jest stosowane do argumentu:

`(W1 50)`  
`50`

Zaczynamy od utworzenia ramki, w której parametr formalny `amount` procedury `W1` jest powiązany z argumentem 50. Istotną rzeczą, którą należy zauważyć,



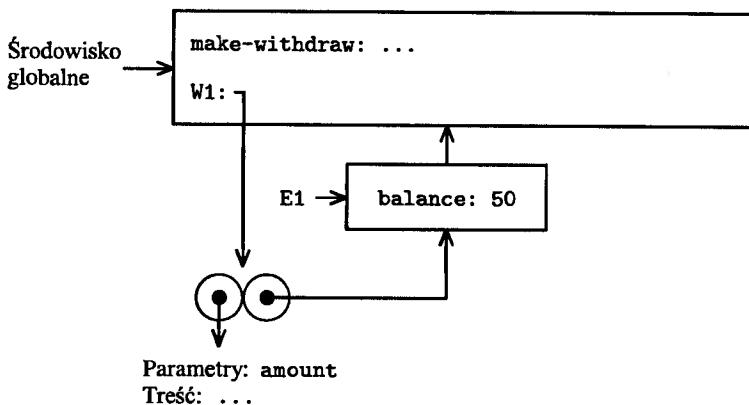
Rys. 3.8. Środowiska utworzone przez zastosowanie obiektu proceduralnego `W1`

jest to, że środowiskiem otaczającym tę ramkę nie jest środowisko globalne, lecz środowisko `E1`, gdyż właśnie to środowisko jest wskazywane przez obiekt proceduralny `W1`. Wewnątrz tego nowego środowiska obliczamy treść procedury:

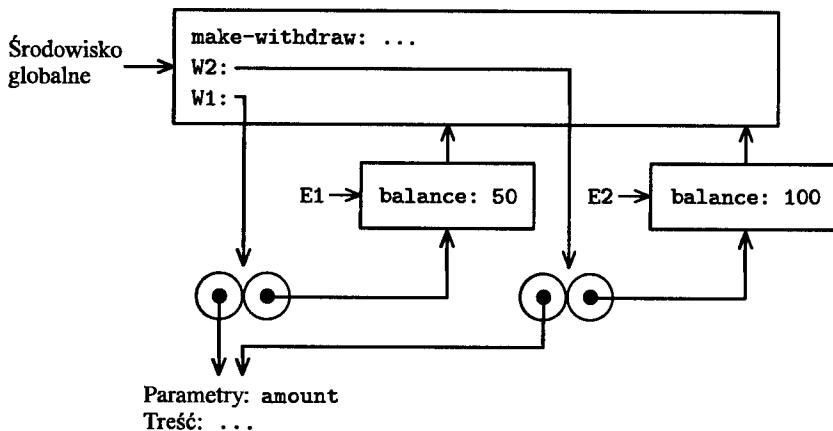
```
(if (>= balance amount)
  (begin (set! balance (- balance amount))
        balance)
  "Brak środków na koncie")
```

Uzyskana struktura środowisk jest pokazana na rys. 3.8. Obliczane wyrażenie odwołuje się zarówno do `amount`, jak i `balance`. `Amount` jest odnajdywane w pierwszej ramce środowiska, a `balance` w ramce `E1` wskazywanej przez wskaznik do środowiska otaczającego.

Kiedy wykonywane jest `set!`, wtedy zmieniane jest wiązanie `balance` w `E1`. W chwili zakończenia wywołania `W1` wartość `balance` wynosi 50, a ramka zawierająca `balance` jest nadal wskazywana przez obiekt proceduralny `W1`. Ramka wiążąca `amount` (w której wykonywaliśmy kod, który zmienił `balance`) nie ma już znaczenia, gdyż wywołanie procedury, które ją utworzyło, zakończyło się i nie ma żadnych wskazników prowadzących do tej ramki z innych części środowiska. Gdy po raz kolejny wywołujemy `W1`, tworzona jest nowa ramka wiążąca `amount`, której środowiskiem otaczającym jest `E1`. Widzimy więc, że `E1` służy jako „miejsce” przechowywania lokalnej zmiennej stanu obiektu proceduralnego `W1`. Na rysunku 3.9 widać sytuację po wywołaniu `W1`.



Rys. 3.9. Środowiska po wywołaniu W1



Rys. 3.10. Wynik utworzenia drugiego obiektu za pomocą (define W2 (make-withdraw 100))

Zobaczmy, co się dzieje, gdy tworzymy drugi „procesor wypłat”, wywołując ponownie make-withdraw:

```
(define W2 (make-withdraw 100))
```

Powstaje wówczas struktura środowisk przedstawiona na rys. 3.10, gdzie widać, że W2 jest obiektem proceduralnym, który jest parą złożoną z pewnego kodu i środowiska. Środowisko E2 zostało utworzone dla W2 przez wywołanie procedury make-withdraw. Zawiera ono ramkę, w której znajduje się wiązanie lokalne zmiennej balance. Jednakże W1 i W2 mają ten sam kod — określony przez lambda-wyrażenie z treści make-withdraw<sup>15</sup>. Widać teraz, dlaczego

<sup>15</sup> To, czy W1 i W2 współdzielą ten sam fizyczny kod pamiętany w komputerze, czy też każde z nich trzyma własną kopię kodu, stanowi szczegół zależny od implementacji. W przypadku interpretera, który implementujemy w rozdziale 4, kod jest rzeczywiście współdzielony.

W1 i W2 zachowują się jak niezależne obiekty. Wywołania W1 odwołują się do zmiennej stanu balance przechowywanej w E1, podczas gdy wywołania W2 odwołują się do balance przechowywanej w E2. Tak więc zmiany lokalnego stanu jednego obiektu nie wpływają na stan drugiego.

### Ćwiczenie 3.10

W procedurze `make-withdraw` zmienna lokalna `balance` powstaje jako parametr procedury `make-withdraw`. Moglibyśmy również utworzyć taką lokalną zmienną stanu za pomocą `let` w następujący sposób:

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Brak środków na koncie"))))
```

Przypomnijmy, że w punkcie 1.3.2 stwierdziliśmy, iż `let` jest tylko lukrem syntaktycznym pokrywającym wywołanie procedury:

```
(let ((zmienna) wyrażenie)) treść)
```

jest interpretowane jako alternatywna składnia dla

```
((lambda ((zmienna)) treść) wyrażenie)
```

Użyj modelu środowiskowego do zanalizowania tej alternatywnej wersji `make-withdraw` i wykonaj rysunki, podobne do tych w tekście, ilustrujące następujące wywołania:

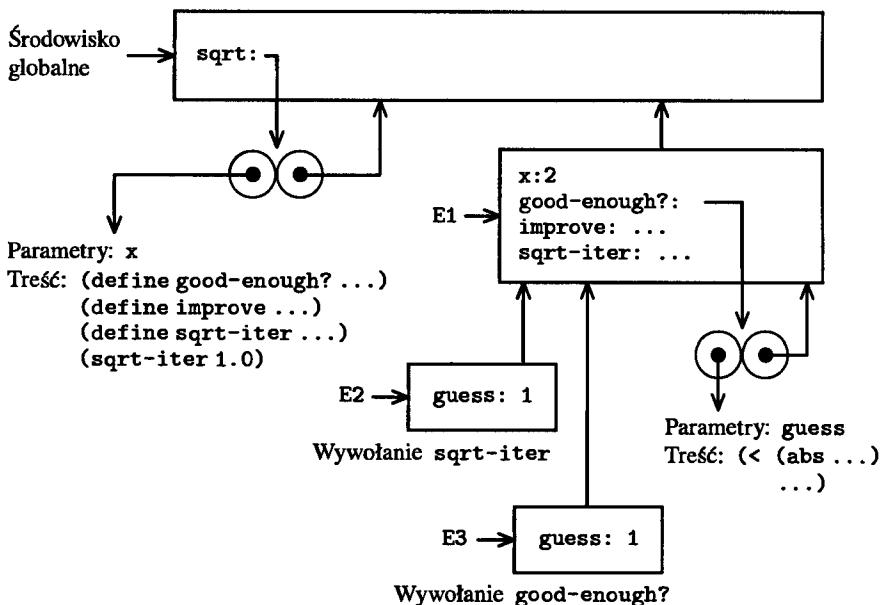
```
(define W1 (make-withdraw 100))
(W1 50)
(define W2 (make-withdraw 100))
```

Pokaż, że obie wersje `make-withdraw` tworzą tak samo zachowujące się obiekty. Czym się różnią struktury środowisk w przypadku tych dwóch wersji?

### 3.2.4. Definicje wewnętrzne

W punkcie 1.1.8 przedstawiliśmy koncepcję polegającą na tym, że procedury mogą zawierać definicje wewnętrzne, co prowadzi do powstania struktury blokowej, takiej jak w poniższym programie obliczającym pierwiastki kwadratowe:

```
(define (sqrt x)
  (define (good-enough? guess)
```



Rys. 3.11. Procedura `sqrt` z definicjami wewnętrznyimi

```
(< (abs (- (square guess) x)) 0.001))
(define (improve guess)
  (average guess (/ x guess)))
(define (sqrt-iter guess)
  (if (good-enough? guess)
      guess
      (sqrt-iter (improve guess))))
(sqrt-iter 1.0))
```

Możemy teraz zastosować model środowiskowy i zobaczyć, dlaczego definicje wewnętrzne działają tak jak powinny. Na rysunku 3.11 jest przedstawiona sytuacja, w której podczas obliczania wyrażenia (`sqrt 2`) jest po raz pierwszy wywoływana procedura `good-enough?` z parametrem `guess` równym 1.

Zwrócić uwagę na strukturę środowiska. `Sqrt` to symbol powiązany w środowisku globalnym z obiektem proceduralnym odwołującym się do środowiska globalnego. Gdy wywołujemy `sqrt`, powstaje nowe środowisko `E1` (pod względem środowiska globalnego), w którym parametr `x` jest związany z liczbą 2. Treść `sqrt` jest obliczana w środowisku `E1`. Ponieważ pierwszym wyrażeniem w treści `sqrt` jest

```
(define (good-enough? guess)
  (< (abs (- (square guess) x)) 0.001))
```

więc obliczenie tego wyrażenia definiuje w środowisku E1 procedurę `good-enough?`. Mówiąc bardziej ściśle, do pierwszej ramki środowiska E1 jest dodawany symbol `good-enough?` powiązany z obiektem proceduralnym odwołującym się do środowiska E1. Podobnie, `improve` i `sqrt-iter` są zdefiniowane jako procedury w środowisku E1. Dla zwięzłości, na rys. 3.11 jest przedstawiony tylko obiekt proceduralny `good-enough?`.

Po zdefiniowaniu procedur lokalnych jest obliczane wyrażenie (`sqrt-iter 1.0`), nadal w środowisku E1. Na skutek tego obiekt proceduralny związany w środowisku E1 z symbolem `sqrt-iter` jest wywoływany z argumentem 1. Powstaje wówczas środowisko E2, w którym `guess` (parametr procedury `sqrt-iter`) jest powiązany z liczbą 1. Z kolei `sqrt-iter` wywołuje `good-enough?`, przekazując jej jako argument wartość `guess` (określoną w E2). Tworzony jest wówczas kolejne środowisko, E3, w którym `guess` (parametr procedury `good-enough?`) jest powiązany z liczbą 1. Pomimo że zarówno `sqrt-iter`, jak i `good-enough?` mają parametry o nazwie `guess`, są to dwie różne zmienne lokalne znajdujące się w różnych ramkach. Ponadto E1 jest środowiskiem otaczającym zarówno E2, jak i E3, ponieważ obie procedury, `sqrt-iter` i `good-enough?`, w swojej środowiskowej części zawierają odwołanie do E1. Jedną z konsekwencji tego jest to, że symbol `x`, który pojawia się w treści `good-enough?`, odnosi się do wiązania `x` określonego w E1, a dokładniej do wartości `x`, z którą wywołano początkowo procedurę `sqrt`.

Model środowiskowy wyjaśnia więc dwie zasadnicze własności, które powodują, że wykorzystanie lokalnych definicji procedur staje się użyteczną techniką modularyzacji programów:

- Nazwy procedur lokalnych nie kolidują z nazwami procedur zewnętrznych względem otaczającej je procedury, gdyż nazwy procedur lokalnych są wiązane w ramce tworzonej przez tę procedurę w momencie jej wywołania zamiast w środowisku globalnym.
- Procedury lokalne mogą odwoływać się do parametrów otaczających je procedur, używając po prostu nazw tych parametrów jako zmiennych wolnych. Dzieje się tak dlatego, że treść procedury lokalnej jest obliczana w środowisku podziemnym względem środowiska, w którym jest obliczana procedura otaczająca.

### Ćwiczenie 3.11

W punkcie 3.2.3 zobaczyliśmy, jak w modelu środowiskowym można opisywać działanie procedur ze stanem lokalnym. Teraz dowiedzieliśmy się, jak działają definicje wewnętrzne. Typowa procedura przekazująca komunikaty zawiera oba te aspekty. Rozważmy procedurę konta bankowego z punktu 3.1.1:

```
(define (make-account balance)
  (define (withdraw amount)
```

```

(if (>= balance amount)
    (begin (set! balance (- balance amount))
           balance)
    "Brak środków na koncie"))
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(define (dispatch m)
  (cond ((eq? m 'withdraw) withdraw)
        ((eq? m 'deposit) deposit)
        (else (error "Nieznana operacja -- MAKE-ACCOUNT"
                     m))))
dispatch)

```

Przedstaw strukturę środowisk tworzonych przez następujący ciąg wywołań:

```

(define acc (make-account 50))

((acc 'deposit) 40)
90

((acc 'withdraw) 60)
30

```

Gdzie jest przechowywany stan lokalny konta acc? Przypuśćmy, że zdefiniujemy drugie konto:

```
(define acc2 (make-account 100))
```

W jaki sposób stany lokalne obydwu kont są przechowywane osobno? Jaka część struktury środowiska jest współdzielona przez acc i acc2?

### 3.3. Modelowanie z użyciem danych modyfikowalnych

W rozdziale 2 zajmowaliśmy się danymi złożonymi, jako środkami służącymi do budowania obiektów obliczeniowych składających się z wielu części, w celu modelowania obiektów ze świata rzeczywistego mających wiele aspektów. Wprowadziłismy tam zasadę abstrakcji danych, zgodnie z którą struktury danych specyfikuje się za pomocą konstruktorów (tworzących obiekty danych) i selektorów (wydobywających części złożonych obiektów danych). Teraz jednak wiemy, że istnieje jeszcze inny aspekt danych, którego w rozdziale 2 nie poruszyliśmy. Chęć modelowania systemów złożonych z obiektów, które mogą zmieniać swój stan, prowadzi do konieczności modyfikowania złożonych obiektów danych, niezależnie od ich konstruowania i wydobywania ich składowych. Aby móc modelować obiekty złożone, których stan może się zmieniać, wprowadzimy do zasady abstrakcji danych, oprócz selektorów i konstruktorów, operacje nazywane *modyfikatorami* (ang. *mutators*), które zmieniają obiekty danych. Na przykład modelowanie systemu bankowego wymaga modyfikowania

stanu kont (saldo). Tak więc struktura danych przeznaczona do reprezentowania kont bankowych mogłaby dopuszczać operację

`(set-balance! <konto> <nowa wartość>)`

która zmienia saldo określonego konta na określoną nową wartość. Obiekty danych, dla których są zdefiniowane modyfikatory, są znane jako *modyfikowalne obiekty danych* (ang. *mutable data objects*).

W rozdziale 2 wprowadziliśmy pary jako uniwersalny „klej” umożliwiający tworzenie danych złożonych. Niniejszy podrozdział rozpoczniemy od zdefiniowania podstawowych modyfikatorów par, żeby pary mogły służyć nam jako elementy składowe przy budowaniu modyfikowalnych obiektów danych. Modyfikatory te znacznie zwiększają możliwości reprezentacyjne par, pozwalając nam na tworzenie struktur danych innych niż ciągi i drzewa, którymi posługiwaliśmy się w podrozdziale 2.2. Przedstawimy również kilka przykładowych symulacji, w których systemy złożone są modelowane jako kolekcje obiektów ze stanami lokalnymi.

### 3.3.1. Modyfikowalne struktury listowe

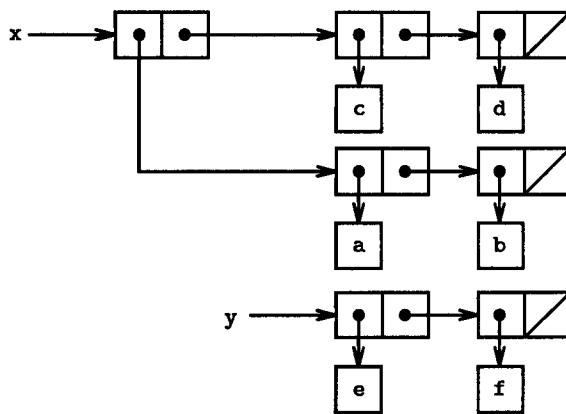
Za pomocą podstawowych operacji na parach — `cons`, `car` i `cdr` — możemy tworzyć struktury listowe i wydobywać ich składowe, ale nie możemy ich modyfikować. To samo dotyczy używanych dotychczas operacji na listach, takich jak `append` czy `list`, gdyż można je zdefiniować za pomocą `cons`, `car` i `cdr`. Do modyfikowania struktur listowych potrzebujemy nowych operacji.

Pierwotne modyfikatory par to `set-car!` i `set-cdr!`. `Set-car!` ma dwa argumenty, z których pierwszy musi być parą. Modyfikuje on tę parę, zastępując wskaźnik `car` wskaźnikiem do drugiego swojego argumentu<sup>16</sup>.

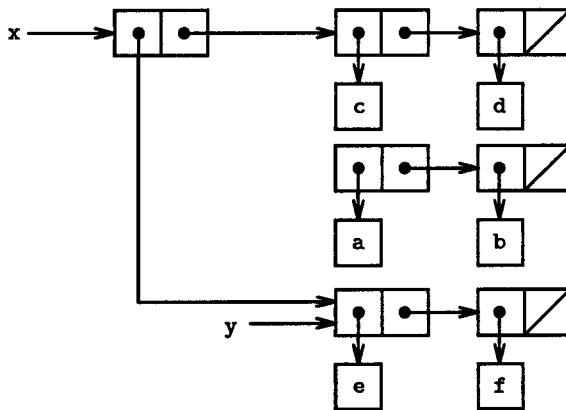
Przypuśćmy na przykład, że `x` jest powiązane z listą `((a b) c d)`, a `y` z listą `(e f)`, jak widać to na rys. 3.12. Obliczenie wyrażenia `(set-car! x y)` modyfikuje parę, z którą jest związany symbol `x`, zastępując jej `car` wartością `y`. Wynik tej operacji jest przedstawiony na rys. 3.13. Struktura `x` została zmieniona i interpreter wypisałby ją jako `((e f) c d)`. Pary reprezentujące listę `(a b)`, określone przez wskaźnik, który został zmieniony, są odłączone od początkowej struktury<sup>17</sup>.

<sup>16</sup> Wartości będące wynikami `set-car!` i `set-cdr!` zależą od implementacji. Podobnie jak w przypadku `set!`, powinniśmy ich używać tak, aby polegać jedynie na efektych ich obliczania, a nie na wartościach będących ich wynikami.

<sup>17</sup> Widzimy więc, że operacje modyfikujące listy mogą wytwarzać bezużyteczne dane, tzw. „śmieci”, które nie są częścią żadnej dostępnej struktury danych. W punkcie 5.3.2 zobaczymy, że system zarządzania pamięcią w Lispie zawiera *program odśmiecający pamięć* (ang. *garbage collector*), który rozpoznaje niepotrzebne pary i zwalnia zajmowaną przez nie pamięć.



Rys. 3.12. Listy x: ((a b) c d) i y: (e f)

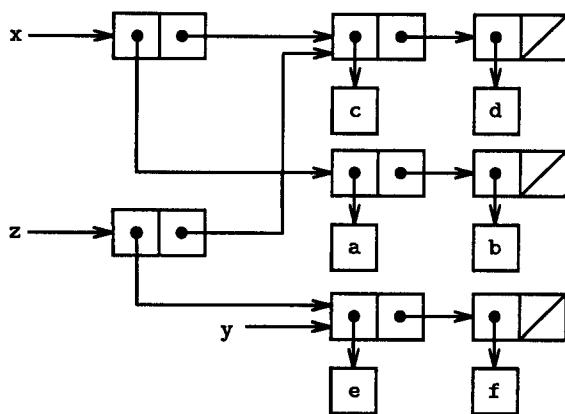


Rys. 3.13. Wynik wykonania (set-car! x y) na listach z rys. 3.12

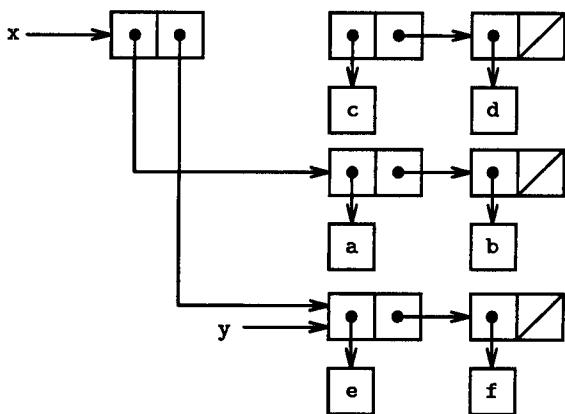
Porównajmy rys. 3.13 z rys. 3.14, który ilustruje wynik wykonania (define z (cons y (cdr x))) dla symboli x i y związań z początkowymi listami z rys. 3.12. Zmienna z jest wiązana z nową parą tworzoną przez operację cons, a lista, z którą związany jest symbol x, pozostaje niezmieniona.

Operacja `set-cdr!` jest analogiczna do `set-car!`. Jedyna różnica polega na tym, że zmieniany jest wskaźnik `cdr` pary zamiast `car`. Na rysunku 3.15 jest pokazany wynik wykonania (`set-cdr! x y`) na listach z rys. 3.12. Wskaźnik `cdr` argumentu `x` został tutaj zastąpiony przez wskaźnik do (`e f`). Ponadto lista (`c d`), która stanowiła `cdr` argumentu `x`, została oddzielona od struktury danych.

Operacja `cons` buduje nową strukturę listową, tworząc nowe pary, podczas gdy `set-car!` i `set-cdr!` modyfikują istniejące pary. W rzeczywistości moglibyśmy zaimplementować `cons` za pomocą tych dwóch modyfikatorów oraz



Rys. 3.14. Wynik wykonania (define z (cons y (cdr x))) na listach z rys. 3.12



Rys. 3.15. Wynik wykonania (set-cdr! x y) na listach z rys. 3.12

procedury `get-new-pair`, której wynikiem jest nowa para nie będąca częścią żadnej istniejącej struktury listowej. Tworzymy nową parę, ustawiamy jej wskaźniki `car` i `cdr` na dane obiekty i przekazujemy nową parę jako wynik `cons`<sup>18</sup>.

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

<sup>18</sup> Operacja `get-new-pair` jest jedną z operacji, które muszą być zaimplementowane jako część systemu zarządzania pamięcią, potrzebnego do zaimplementowania Lispu. Kwestię tę omawiamy w punkcie 5.3.1.

**Ćwiczenie 3.12**

W punkcie 2.2.1 wprowadziliśmy następującą procedurę dodającą elementy na końcu listy:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

Append tworzy nową listę, dokładając kolejno za pomocą cons elementy listy x do listy y. Procedura append! jest podobna do append, ale jest ona modyfikatorem, a nie konstruktorem. Łączy ona listy przez sklejenie ich razem, modyfikując przy tym ostatnią parę listy x tak, aby jej cdr był równy y. (Wywołanie append! z pustą listą x jest błędem).

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

Wynikiem użytej powyżej procedury last-pair jest ostatnia para jej argumentu:

```
(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))
```

Rozważmy następujący ciąg wywołań:

```
(define x (list 'a 'b))
(define y (list 'c 'd))
(define z (append x y))
z
(a b c d)
(cdr x)
<odpowiedź systemu>
(define w (append! x y))
w
(a b c d)
(cdr x)
<odpowiedź systemu>
```

Jakie są brakujące *<odpowiedź systemu>*? Narysuj diagramy pudelkowo-wskaźnikowe wyjaśniające odpowiedź.

**Ćwiczenie 3.13**

Rozważmy następującą procedurę `make-cycle`, która korzysta z procedury `last-pair` zdefiniowanej w ćwiczeniu 3.12:

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x))
```

Narysuj diagram pudełkowo-wskaźnikowy ukazujący strukturę listy `z` tworzonej przez

```
(define z (make-cycle (list 'a 'b 'c)))
```

Co się stanie, jeżeli spróbujemy obliczyć `(last-pair z)`?

**Ćwiczenie 3.14**

Poniższa procedura jest całkiem użyteczna, choć zagmatwana:

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '()))
```

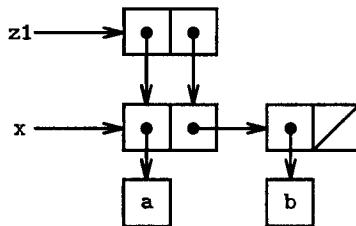
Procedura `loop` używa „tymczasowej” zmiennej `temp` do przechowywania starej wartości `cdr x`, gdyż w następnym wierszu operacja `set-cdr!` niszczy `cdr`. Wyjaśnij, co procedura `mystery` robi w ogólnym przypadku. Przypuśćmy, że `v` jest zdefiniowane jako `(define v (list 'a 'b 'c 'd))`. Narysuj diagram pudełkowo-wskaźnikowy przedstawiający listę związaną z `v`. Przypuśćmy, że obliczamy wyrażenie `(define w (mystery v))`. Narysuj diagram pudełkowo-wskaźnikowy ukazujący stany struktur `v` i `w` po obliczeniu tego wyrażenia. Co wypisałby system jako wartości `v` i `w`?

**Współdzielenie i tożsamość**

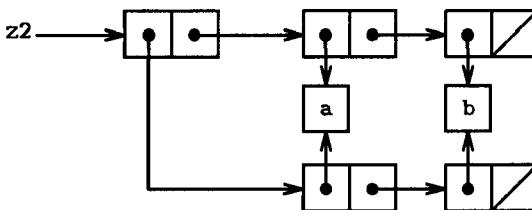
W punkcie 3.1.3 napomknęliśmy o teoretycznych kwestiach „niezmienności” i „zmiany”, wynikających z wprowadzenia przypisania. Kwestie te pojawiają się w praktyce z chwilą, gdy te same pary są *współdzielone* (wspólnie wykorzystywane) przez różne obiekty danych. Rozważmy na przykład strukturę tworzoną przez

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

Jak widać na rys. 3.16, `z1` jest parą, której zarówno `car`, jak i `cdr` wskazują na tę samą parę `x`. Takie współdzielenie pary `x` przez `car` i `cdr` pary `z1` wynika ze sposobu zaimplementowania `cons` w prost. Na ogólnie, budując listy



Rys. 3.16. Lista z1 utworzona przez (cons x x)



Rys. 3.17. Lista z2 utworzona przez (cons (list 'a 'b) (list 'a 'b))

za pomocą cons, uzyskamy wzajemnie powiązane struktury par, w których wiele poszczególnych par jest współdzielonych przez wiele różnych struktur.

W odróżnieniu od rys. 3.16, rys. 3.17 ukazuje strukturę utworzoną przez

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

W strukturze tej pary tworzące dwie listy (a b) są różne, chociaż same symbole są współdzielone<sup>19</sup>.

Gdy myślimy o listach, zarówno z1, jak i z2 reprezentują „takie same” listy, ((a b) a b). Na ogół współdzielenie jest zupełnie niezauważalne, jeżeli operujemy na listach wyłącznie za pomocą cons, car i cdr. Jeżeli jednak dopuścimy używanie modyfikatorów struktur listowych, to współdzielenie nabiera znaczenia. Znaczenie współdzielenia możemy prześledzić na przykładzie następującej procedury, która zmienia car struktury będącej jej argumentem:

```
(define (set-to-wow! x)
  (set-car! (car x) 'wow)
  x)
```

Pomimo że z1 i z2 są „takimi samymi” strukturami, zastosowanie do nich set-to-wow! daje różne wyniki. W przypadku z1 zmiana car oznacza rów-

<sup>19</sup> Te dwie pary są różne, ponieważ każde wywołanie cons tworzy nową parę. Symbole są współdzielone; w języku Scheme dla każdej zadanej nazwy istnieje dokładnie jeden niepowtarzalny symbol o tej nazwie. Ponieważ Scheme nie zapewnia możliwości modyfikowania symboli, takie współdzielenie jest niewykrywalne. Zauważmy również, że współdzielenie sprawdza porównywanie symboli za pomocą eq? do prostego porównywania wskaźników.

nież zmianę cdr, gdyż w z1 car i cdr stanowią tę samą parę. Natomiast w przypadku z2 car i cdr są różne, więc set-to-wow! zmienia tylko car:

```
z1
((a b) a b)

(set-to-wow! z1)
((wow b) wow b)

z2
((a b) a b)

(set-to-wow! z2)
((wow b) a b)
```

Jeden ze sposobów wykrywania współdzielienia struktur listowych polega na użyciu predykatu `eq?`, wprowadzonego przez nas w punkcie 2.3.1 jako metoda porównywania symboli. W ogólności, `(eq? x y)` sprawdza, czy x i y to te same obiekty (tzn. czy x i y są równe jako wskaźniki). Tak więc dla z1 i z2, zdefiniowanych jak na rys. 3.16 i 3.17, `(eq? (car z1) (cdr z1))` jest prawdą, a `(eq? (car z2) (cdr z2))` jest fałszem.

Jak zobaczymy w kolejnych punktach, możemy wykorzystać współdzielienie danych do znacznego rozszerzenia repertuaru struktur danych, które można reprezentować za pomocą par. Jednakże współdzielanie może być również niebezpieczne, gdyż zmiany dokonane w jednej strukturze danych wpływają także na stan innych struktur, które akurat współdzielą zmieniane części struktury. Operacji `set-car!` i `set-cdr!` należy używać ostrożnie — jeżeli nie zdajemy sobie dokładnie sprawy z tego, jak nasze obiekty danych są współdzielone, to modyfikowanie ich może mieć nieprzewidziane skutki<sup>20</sup>.

### Ćwiczenie 3.15

Narysuj diagramy pudełkowo-wskaźnikowe wyjaśniające wynik zastosowania procedury `set-to-wow!` do obiektów z1 i z2 przedstawionych powyżej.

### Ćwiczenie 3.16

Ben Bajerbit postanowił napisać procedurę obliczającą liczbę par tworzących dowolną strukturę listową. „To przecież proste” — rozumieje. „Liczba par tworzących dowolną

<sup>20</sup> Subtelności związane z operowaniem modyfikowalnymi obiektami współdzielącymi dane odzwierciedlają kryjące się za tym kwestie „niezmienności” i „zmiany”, o których wspominaliśmy w punkcie 3.1.3. Stwierdziliśmy wówczas, że wprowadzenie do naszego języka możliwości modyfikowania obiektów wymaga, aby obiekty miały „tożsamość”, która jest czymś innym niż sumą części składających się na obiekt. W Lispie przyjmujemy, że „tożsamość” jest cechą, którą można badać za pomocą predykatu `eq?`, tzn. za pomocą równości wskaźników. Ponieważ w większości implementacji Lispu wskaźnik jest w gruncie rzeczy adresem pamięci, tym samym „rozwiązujeśmy problem” definicji tożsamości obiektów, przyjmując, że obiekt danych „stanowi” informację przechowywaną w określonych komórkach pamięci komputera. Podejście takie jest wystarczające na potrzeby prostych programów w Lispie, ale raczej nie wystarcza do ogólnego rozwiązania problemu „tożsamości” w modelach obliczeniowych.

strukturę jest równa liczbie par tworzących jej `car` plus liczba par tworzących jej `cdr` plus jedna (bieżąca) para". Tak więc Ben napisał następującą procedurę:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
          (count-pairs (cdr x))
          1)))
```

Pokaż, że ta procedura jest niepoprawna. W szczególności, narysuj diagram pudełkowo-wskaźnikowy przedstawiający struktury listowe zbudowane z dokładnie trzech par, dla których procedura Bena da w wyniku: 3; 4; 7; wynik nigdy nie zostanie obliczony.

### Ćwiczenie 3.17

Opracuj poprawną wersję procedury `count-pairs` z ćwiczenia 3.16, której wynikiem jest liczba różnych par tworzących daną strukturę. (Wskazówka: Przejrzyj daną strukturę, używając dodatkowej struktury danych do śledzenia, które pary zostały już policzone).

### Ćwiczenie 3.18

Napisz procedurę, która bada listę i stwierdza, czy zawiera ona cykl, tzn. stwierdza, czy program, który stara się znaleźć koniec listy, przechodząc jej kolejne `cdr`y, się zapętli. Taka lista powstała na przykład w ćwiczeniu 3.13.

### Ćwiczenie 3.19

Rozwiąż ponownie ćwiczenie 3.18, używając algorytmu działającego w stałej pamięci [i liniowym czasie; przyp. tłum.]. (Wymaga to dużo sprytu i pomysłowości).

### Modyfikacja to tylko przypisanie

Gdy w punkcie 2.1.3 wprowadziliśmy dane złożone, zauważliśmy, że pary mogą również być reprezentowane wyłącznie za pomocą procedur:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Niezdefiniowana operacja -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))
```

To samo stwierdzenie jest prawdziwe dla danych modyfikowalnych. Modyfikowalne obiekty danych można zaimplementować jako procedury za pomocą przypisania i stanu lokalnego: Możemy na przykład rozszerzyć powyższą implementację par o operacje `set-car!` i `set-cdr!` w sposób analogiczny do

tego, jak zaimplementowaliśmy w punkcie 3.1.1 konta bankowe za pomocą procedury `make-account`:

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Niezdefiniowana operacja -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))

(define (set-car! z new-value)
  ((z 'set-car!) new-value)
  z)

(define (set-cdr! z new-value)
  ((z 'set-cdr!) new-value)
  z)
```

Przypisanie to wszystko, co teoretycznie jest potrzebne do określenia zachowania danych modyfikowalnych. Gdy tylko wprowadzamy do naszego języka operację `set!`, poruszamy tym samym wszystkie kwestie związane nie tylko z przypisaniem, ale ogólnie z danymi modyfikowalnymi<sup>21</sup>.

### Ćwiczenie 3.20

Narysuj diagramy środowisk przedstawiające obliczenie poniższego ciągu wyrażeń, używając przedstawionej powyżej proceduralnej implementacji par:

```
(define x (cons 1 2))
(define z (cons x x))
(set-car! (cdr z) 17)

(car x)
17
```

(Porównaj ćwiczenie 3.11).

<sup>21</sup> Z kolei z punktu widzenia implementacji przypisanie wymaga dokonania zmiany w środowisku, które samo jest modyfikowalną strukturą danych. Tak więc przypisanie i modyfikowanie struktur danych są równoważne — każde z nich może być zaimplementowane za pomocą drugiego.

### 3.3.2. Reprezentowanie kolejek

Modyfikatory `set-car!` i `set-cdr!` pozwalają nam na budowanie za pomocą par struktur danych, których nie moglibyśmy zbudować, korzystając wyłącznie z `cons`, `car` i `cdr`. W niniejszym punkcie pokazujemy, jak można za pomocą par reprezentować strukturę danych nazywaną kolejką. W punkcie 3.3.3 pokażemy, jak można reprezentować struktury danych nazywane tablicami.

*Kolejka* (ang. *queue*) to ciąg, do którego elementy są wstawiane z jednej strony (nazywanej *końcem kolejki*), a usuwane z drugiej strony (nazywanej *początkiem kolejki*). Na rysunku 3.18 widać początkowo pustą kolejkę, do której są wstawiane elementy `a` i `b`. Następnie `a` jest usuwane, wstawiane są `c` i `d`, po czym usuwane jest `b`. Ponieważ elementy są zawsze usuwane w tej samej kolejności, w której są wstawiane, kolejka jest czasami nazywana strukturą *FIFO* (pierwszy na wejściu, pierwszy na wyjściu; ang. *first in, first out*).

Zgodnie z zasadą abstrakcji danych możemy patrzeć na kolejki jak na zdefiniowane za pomocą następującego zestawu operacji:

- konstruktor:

`(make-queue)`

daje w wyniku pustą kolejkę (tzn. kolejkę nie zawierającą żadnych elementów);

- dwa selektory:

`(empty-queue? <kolejka>)`

sprawdza, czy kolejka jest pusta;

`(front-queue <kolejka>)`

daje w wyniku obiekt stojący na początku kolejki; jeśli kolejka jest pusta, to zgłoszany jest błąd; operacja ta nie modyfikuje kolejki;

- dwa modyfikatory:

`(insert-queue! <kolejka> <element>)`

wstawia element na koniec kolejki; wynikiem jest tak zmodyfikowana kolejka;

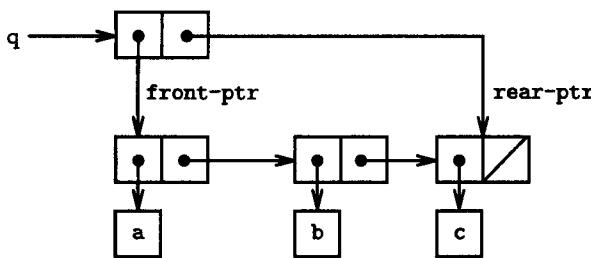
`(delete-queue! <kolejka>)`

usuwa element stojący na początku kolejki; wynikiem jest tak zmodyfikowana kolejka; jeśli kolejka (przed usunięciem elementu) jest pusta, to zgłoszany jest błąd.

Ponieważ kolejka jest ciągiem elementów, moglibyśmy ją oczywiście reprezentować jako zwykłą listę — początek kolejki byłby wówczas równy `car` listy, wstawienie elementu na koniec kolejki sprowadzałoby się do dołączenia nowego elementu na końcu listy, a usunięcie elementu z kolejki polegałoby jedynie na obliczeniu `cdr` listy. Jednakże taka reprezentacja nie jest efektywna, gdyż chcąc wstawić element, musimy przejrzeć całą listę, aż dojdziemy do jej końca. Ponieważ jedyną metodą przeglądania listy, jaką znamy, jest wielokrotne wykonywanie operacji `cdr`, takie przejrzenie listy wymaga  $\Theta(n)$  kroków

Operacja	Kolejka wynikowa
(define q (make-queue))	
(insert-queue! q 'a)	a
(insert-queue! q 'b)	a b
(delete-queue! q)	b
(insert-queue! q 'c)	b c
(insert-queue! q 'd)	b c d
(delete-queue! q)	c d

Rys. 3.18. Operacje na kolejce



Rys. 3.19. Implementacja kolejki jako listy ze wskaźnikami do jej początku i końca

dla listy złożonej z  $n$  elementów. Prosta modyfikacja reprezentacji list rozwiązuje ten problem i umożliwia zaimplementowanie operacji na kolejkach tak, że wymagają one  $\Theta(1)$  kroków — tzn. że liczba wykonywanych kroków nie zależy od długości kolejki.

Trudności z reprezentacją listową wynikają z konieczności przeglądania całej listy w celu znalezienia jej końca. Konieczność ta jest spowodowana tym, że choć standardowy sposób reprezentowania list jako łańcucha par z łatwością udostępnia nam wskaźnik do początku listy, jednak nie udostępnia nam żadnego łatwo dostępnego wskaźnika do jej końca. Modyfikacja pozwalająca uniknąć opisanej wady polega na reprezentowaniu kolejki w postaci listy wraz z dodatkowym wskaźnikiem do ostatniej pary na liście. W ten sposób, gdy chcemy wstawić element, możemy sprawdzić ten wskaźnik i uniknąć przeglądania listy.

Kolejkę reprezentujemy jako parę wskaźników `front-ptr` i `rear-ptr`, które wskazują, odpowiednio, na pierwszą i ostatnią parę na zwykłej liście. Ponieważ chcielibyśmy, żeby kolejka była obiektem identyfikowalnym (mającym tożsamość), możemy połączyć obydwa wskaźniki za pomocą `cons`. Tak więc sama kolejka będzie stanowiła `cons` dwóch wskaźników. Na rysunku 3.19 widać taką reprezentację.

Aby zdefiniować operacje na kolejkach, używamy następujących procedur, które pozwalają na wybieranie i modyfikowanie wskaźników do początku i końca kolejki:

```
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
```

Teraz możemy zaimplementować faktyczne operacje na kolejkach. Przyjmujemy, że kolejka jest pusta, jeśli wskaźnik do jej początku jest pustą listą:

```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

Wynikiem konstruktora make-queue jest para, której car i cdr są pustymi listami, reprezentująca początkowo pustą kolejkę:

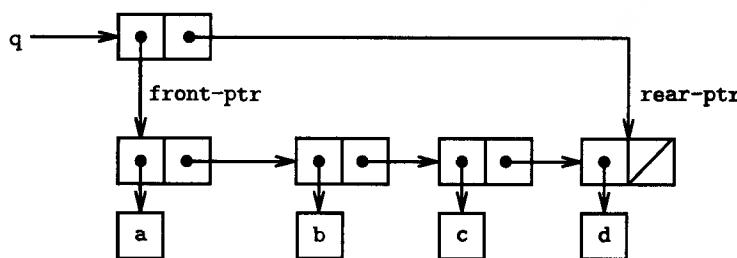
```
(define (make-queue) (cons '() '()))
```

Aby wybrać element znajdujący się na początku kolejki, bierzemy car pary wskazywanej przez wskaźnik do początku kolejki:

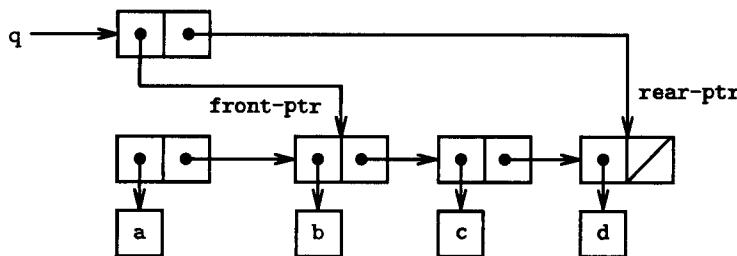
```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "Wywołanie FRONT dla pustej kolejki" queue)
      (car (front-ptr queue))))
```

Aby wstawić element do kolejki, postępujemy zgodnie z metodą, której wynik jest przedstawiony na rys. 3.20. Najpierw tworzymy nową parę, której car jest wstawianym elementem i której cdr jest pustą listą. Jeśli kolejka była początkowo pusta, to ustawiamy wskaźniki do początku i końca kolejki na tę nową parę. W przeciwnym razie modyfikujemy ostatnią parę w kolejce tak, żeby wskazywała na nową parę, oraz ustawiamy wskaźnik do końca kolejki na nową parę.

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
            (set-cdr! (rear-ptr queue) new-pair)
            (set-rear-ptr! queue new-pair)
            queue))))
```



Rys. 3.20. Wynik zastosowania (`insert-queue! q 'd`) do kolejki przedstawionej na rys. 3.19



Rys. 3.21. Wynik zastosowania (`delete-queue! q`) do kolejki przedstawionej na rys. 3.20

Aby usunąć element z początku kolejki, wystarczy jedynie zmienić wskaźnik do początku kolejki tak, żeby wskazywał na jej drugi element, który jest wskazywany przez `cdr` pierwszego elementu (zob. rys. 3.21)<sup>22</sup>:

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "Wywołanie DELETE! dla pustej kolejki" queue))
        (else
          (set-front-ptr! queue (cdr (front-ptr queue)))
          queue)))
```

### Ćwiczenie 3.21

Ben Bajerbit postanowił przetestować opisaną powyżej implementację kolejek. Wprowadził procedury do interpretera Lispu i przystąpił do ich wypróbowywania:

```
(define q1 (make-queue))
(insert-queue! q1 'a)
((a) a)
```

<sup>22</sup> Jeśli pierwszy element jest jednocześnie ostatnim elementem kolejki, to po usunięciu go wskaźnik do początku kolejki będzie pustą listą, co oznacza, że kolejka będzie pusta; nie musimy się martwić o wskaźnik do jej końca, który nadal będzie wskazywał na usunięty element, gdyż `empty-queue?` sprawdza jedynie wskaźnik do początku kolejki.

```
(insert-queue! q1 'b)
((a b) b)

(delete-queue! q1)
((b) b)

(delete-queue! q1)
(() b)
```

„Wszystko jest źle!” — narzeka Ben. „Odpowiedzi interpretera wskazują, że ostatni element jest wstawiany do kolejki dwukrotnie. A gdy usuwam obydwa elementy, drugie b nadal jest w kolejce, a więc nie jest ona pusta, choć powinna być”. Ewa Lu Ator sugeruje delikatnie Benowi, że źle zrozumiał, co się dzieje. „Elementy nie są wstawiane do kolejki dwukrotnie” — wyjaśnia Ewa. „To tylko standardowy program wypisujący Lispu nie wie, jak interpretować reprezentację kolejki. Jeśli chcesz, aby kolejki były wypisywane poprawnie, musisz zdefiniować własną procedurę wypisywania kolejek”. Wyjaśnij, o czym mówi Ewa. W szczególności wyjaśnij, dlaczego przykłady Ben'a dają takie wyniki, a nie inne. Zdefiniuj procedurę `print-queue`, której argumentem jest kolejka i która wypisuje ciąg elementów należących do tej kolejki.

### Ćwiczenie 3.22

Zamiast reprezentować kolejkę jako parę wskaźników, możemy reprezentować ją jako procedurę ze stanem lokalnym. Stan lokalny będzie się składał ze wskaźników do początku i końca zwykłej listy. Tak więc procedura `make-queue` będzie miała następującą postać:

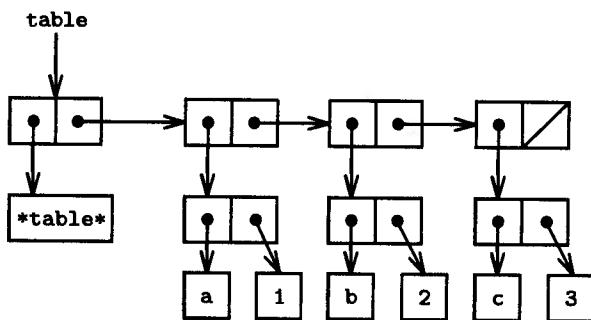
```
(define (make-queue)
  (let ((front_ptr ... ))
    (rear_ptr ... ))
  (definicje procedur wewnętrznych)
  (define (dispatch m) ...)
  dispatch))
```

Uzupełnij definicję `make-queue` i podaj implementacje operacji na kolejkach używających tej reprezentacji.

### Ćwiczenie 3.23

*Kolejka dwukierunkowa* (ang. *deque* = *double-ended queue*) to ciąg, do którego elementy mogą być wstawiane i usuwane zarówno na końcu, jak i na początku. Operacje na kolejkach dwukierunkowych to: konstruktor `make-deque`, predykat `empty-deque?`, selektory `front-deque` (początek) i `rear-deque` (koniec) oraz modyfikatory `front-insert-deque!` (wstaw na początek), `rear-insert-deque!` (wstaw na koniec), `front-delete-deque!` (usuń z początku) i `rear-delete-deque!` (usuń z końca). Pokaż, jak można reprezentować kolejki dwukierunkowe za pomocą par, i podaj implementacje operacji na nich<sup>23</sup>. Wszystkie operacje powinny być zrealizowane w  $\Theta(1)$  krokach.

<sup>23</sup> Uważaj, aby interpreter nie starał się wypisywać struktur zawierających cykle. (Zobacz ćwiczenie 3.13).



Rys. 3.22. Reprezentacja tablicy w postaci listy z atrapą

### 3.3.3. Reprezentowanie tablic

Gdy w rozdziale 2 badaliśmy różne sposoby reprezentowania zbiorów, wspomnialiśmy w punkcie 2.3.3 o problemie przechowywania tablicy rekordów indeksowanych identyfikującymi je kluczami. W implementacji programowania sterowanego danymi, w punkcie 2.4.3, w istotny sposób używaliśmy tablic dwuwymiarowych, w których informacja jest zapamiętywana i z których jest odczytywana za pomocą dwóch kluczy. Zobaczmy teraz, jak tworzyć tablice będące modyfikowalnymi strukturami listowymi.

Rozważmy najpierw tablice jednowymiarowe, w których każda wartość jest indeksowana za pomocą jednego klucza. Implementujemy je jako listy rekordów, z których każdy ma postać pary składającej się z klucza i związanej z nim wartości. Rekordy te są sklejone w listę za pomocą par, których cary wskazują na kolejne rekordy. Te sklejające pary są nazywane *szkieletem* (ang. *backbone*) tablicy. Aby mieć do dyspozycji miejsce w pamięci, którego zawartość można zmieniać, gdy dodajemy nowy rekord do tablicy, tworzymy tablicę jako *listę z atrapą* (ang. *headed list*). Szkielet listy z atrapą ma na początku dodatkową parę zawierającą „atraps” — w tym wypadku dowolnie wybrany symbol, na przykład **\*table\***. Na rysunku 3.22 jest przedstawiony diagram pudełkowo-wskaźnikowy tablicy

```

a: 1
b: 2
c: 3
  
```

Aby wydobyć z tablicy informację, używamy procedury `lookup`, której argumentem jest klucz, a wynikiem jest związana z nim wartość (lub fałsz, jeśli takiej wartości nie ma). Procedura `lookup` jest zdefiniowana za pomocą operacji `assoc`, której argumentami są klucz oraz lista rekordów. Zwróćmy uwagę, że `assoc` nigdy nie widzi atrap. Wynikiem `assoc` jest rekord, którego

car jest zadanym kluczem<sup>24</sup>. Następnie lookup sprawdza, czy rekord będący wynikiem assoc nie jest fałszem, i przekazuje go (jego cdr) jako wynik.

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))

(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

Aby wstawić do tablicy wartość indeksowaną za pomocą określonego klucza, wywołujemy najpierw assoc, żeby sprawdzić, czy w tablicy nie ma już rekordu o zadanym kluczu. Jeśli nie ma takiego rekordu, to za pomocą cons tworzony jest nowy rekord zawierający dany klucz i wartość, po czym jest on wstawiany na początek listy rekordów, tuż za atrapą. Jeśli rekord o zadanym kluczu istnieje, to ustawiamy jego cdr na wskazaną nową wartość. Atrapa tablicy stanowi ustalone miejsce w pamięci, którego zawartość jest modyfikowana przy wstawianiu nowych rekordów<sup>25</sup>.

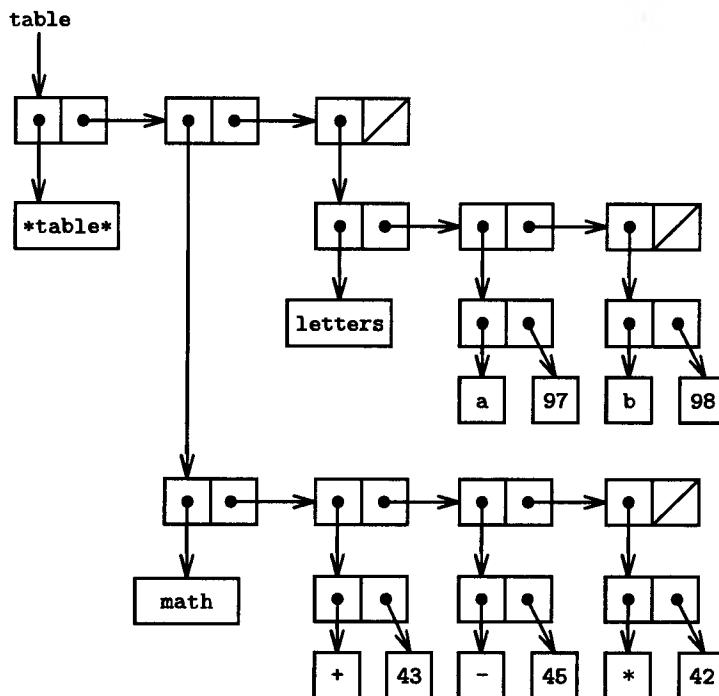
```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))))
  'ok))
```

Aby utworzyć nową tablicę, wystarczy po prostu utworzyć listę zawierającą symbol **\*table\***:

```
(define (make-table)
  (list '*table*))
```

<sup>24</sup> Procedura assoc korzysta z predykatu **equal?**, dzięki czemu rozpoznaje klucze, które mogą być symbolami, liczbami lub strukturami listowymi.

<sup>25</sup> Tak więc pierwsza para szkieletu jest obiektem reprezentującym tablicę; tzn. wskaźnik do tablicy jest wskaźnikiem do tej pary. Właśnie ta para rozpoczyna zawsze tablicę. Gdyby reprezentacja tablicy nie była tak zorganizowana, wynikiem **insert!** w przypadku wstawiania nowego rekordu musiałby być początek zmienionej tablicy.



Rys. 3.23. Tablica dwuwymiarowa

### Tablice dwuwymiarowe

W tablicy dwuwymiarowej każda wartość jest indeksowana za pomocą dwóch kluczy. Możemy zdefiniować taką tablicę jako tablicę jednowymiarową, w której każdy klucz określa podtablicę. Na rysunku 3.23 widać diagram pudełko-wskaźnikowy następującej tablicy zawierającej dwie podtablice:

```

math:
 +: 43
 -: 45
 *: 42
letters:
 a: 97
 b: 98
  
```

(Podtablice nie wymagają dodatkowych atrap, gdyż do tego samego celu może służyć para zawierająca klucz identyfikujący podtablicę).

Gdy szukamy elementu w tablicy, najpierw korzystamy z pierwszego klucza, aby zidentyfikować właściwą podtablicę. Następnie używamy drugiego klucza, aby zidentyfikować rekord w ramach podtablicy.

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              false))
        false)))
```

Aby wstawić do tablicy nowy element indeksowany przez parę kluczów, sprawdzamy za pomocą `assoc`, czy istnieje podtablica odpowiadająca pierwszemu kluczowi. Jeśli nie, to tworzymy nową podtablicę zawierającą jeden rekord (`key-2, value`), którą wstawiamy do tablicy, indeksując ją pierwszym kluczem. Jeśli podtablica odpowiadająca pierwszemu kluczowi już istnieje, to wstawiamy do niej nowy element, używając do tego celu przedstawionej wcześniej procedury wstawiania elementów do tablicy jednowymiarowej:

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                               (cons key-2 value))
                        (cdr table)))))

  'ok)
```

### Tworzenie tablic lokalnych

Tablice, na których operują zdefiniowane powyżej operacje `lookup` i `insert!`, są przekazywane jako argumenty. Pozwala to nam na pisanie programów korzystających z więcej niż jednej tablicy. Inny sposób radzenia sobie z wieloma tablicami polega na tworzeniu oddzielnych procedur `lookup` i `insert!` dla każdej tablicy. Możemy tego dokonać, reprezentując tablicę jako obiekt proceduralny, który zawiera wewnętrzną tablicę będącą częścią jego stanu lokalnego. Gdy taki „obiekt tablicowy” otrzyma stosowny komunikat, dostarcza odpowiedniej procedury, za pomocą której możemy operować na tablicy wewnętrznej. Oto generator tablic dwuwymiarowych reprezentowanych właśnie w ten sposób:

```

(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record
                  (cdr record)
                  false))
            false)))
    (define (insert! key-1 key-2 value)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record
                  (set-cdr! record value)
                  (set-cdr! subtable
                            (cons (cons key-2 value)
                                  (cdr subtable)))))

            (set-cdr! local-table
                      (cons (list key-1
                                  (cons key-2 value))
                            (cdr local-table)))))

      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Nieznanana operacja -- TABLE" m))))
    dispatch))

```

Stosując `make-table`, moglibyśmy w poniższy sposób zaimplementować operacje `get` i `put` używane w punkcie 2.4.3 w programowaniu sterowanym danymi:

```

(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))

```

Argumentami `get` są dwa klucze, a argumentami `put` są dwa klucze i wartość. Obie operacje mają dostęp do tej samej tablicy lokalnej, która jest ukryta wewnątrz obiektu tworzonego przez wywołanie `make-table`.

### Ćwiczenie 3.24

W powyższej implementacji tablic klucze są porównywane za pomocą predykatu `equal?` (wywoływanego przez `assoc`). Nie zawsze jest to właściwy sposób porównywania. Możemy na przykład wyobrazić sobie tablicę, w której klucze są liczbami, ale nie wymagamy, aby szukany klucz był dokładnie równy zadanej liczbie, a jedynie

nie z pewną tolerancją. Opracuj konstruktor tablic `make-table`, którego argumentem jest procedura `same-key?`, używana do badania, czy klucze są „równe”. Wynikiem `make-table` powinna być procedura `dispatch` udostępniająca odpowiednie procedury `lookup` i `insert!` operujące na tablicy lokalnej.

### Ćwiczenie 3.25

Uogólniając jedno- i dwuwymiarowe tablice, pokaż, jak można zaimplementować tablice, w których wartości są indeksowane za pomocą dowolnej liczby kluczy, a różne wartości mogą być indeksowane za pomocą różnej liczby kluczy. Argumentami procedur `lookup` i `insert!` powinny być listy kluczy używane do indeksowania tablicy.

### Ćwiczenie 3.26

Przeszukując tablice zaimplementowane w powyższy sposób, należy przeglądać całą listę rekordów. Jest to w zasadzie reprezentacja za pomocą list nieuporządkowanych, przedstawiona w punkcie 2.3.3. W przypadku dużych tablic bardziej efektywna może być inna struktura danych. Opisz implementację tablic, w której rekordy (klucz, wartość) tworzą drzewo binarne, zakładając, że klucze można w pewien sposób uporządkować (np. numerycznie lub alfabetycznie). (Porównaj ćwiczenie 2.66 z rozdziału 2).

### Ćwiczenie 3.27

*Spamiętywanie* (ang. *memoization*; nazywane również *tablicowaniem* — ang. *tabulation*) to technika umożliwiająca procedurze zapisywanie w lokalnej tablicy wartości wcześniej już obliczonych wyników. Zastosowanie tej techniki może mieć ogromny wpływ na efektywność programów. Procedura spamiętująca utrzymuje tablicę, w której przechowuje wyniki poprzednich wywołań, indeksowane argumentami tych wywołań. Gdy zlećmy takiej procedurze obliczenie wartości, sprawdzi ona najpierw, czy taka wartość nie występuje już w tablicy, i jeśli istnieje, to tylko przekaże tę wartość. W przeciwnym razie obliczy ona w zwykły sposób nową wartość i zapamięta ją w tablicy. Jako przykład spamiętywania przypomnijmy wykładowczy proces obliczania liczb Fibonacciego z punktu 1.2.2:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

Oto wersja tej samej procedury ze spamiętywaniem:

```
(define memo-fib
  (memoize (lambda (n)
              (cond ((= n 0) 0)
                    ((= n 1) 1)
                    (else (+ (memo-fib (- n 1))
                              (memo-fib (- n 2))))))))
```

gdzie procedura `memoize`, realizująca spamiętywanie, jest zdefiniowana następująco:

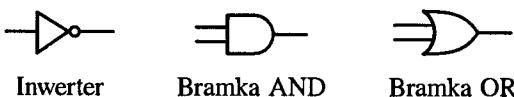
```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

Narysuj diagram środowisk i zanalizuj obliczenie (memo-fib 3). Wyjaśnij, dlaczego memo-fib oblicza  $n$ -tą liczbę Fibonacciego w liczbie kroków proporcjonalnej do  $n$ . Czy mechanizm ten działałby również, gdybyśmy po prostu zdefiniowali memo-fib jako (memoize fib)?

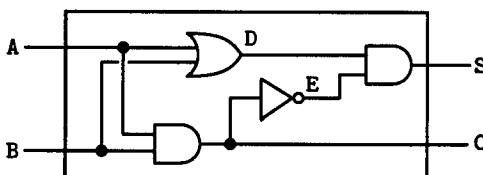
### 3.3.4. Symulator układów cyfrowych

Konstruowanie złożonych systemów cyfrowych, takich jak komputery, to ważna dziedzina inżynierii. Systemy cyfrowe są tworzone przez łączenie elementów prostych. Chociaż zachowanie tych pojedynczych elementów jest proste, sieci złożone z takich elementów charakteryzują się bardzo złożonym zachowaniem. Komputerowa symulacja tworzonych projektów układów stanowi ważne narzędzie pracy inżynierów systemów cyfrowych. W niniejszym punkcie skonstruujemy system symulujący logiczne układy cyfrowe. System ten jest przykładem klasy programów nazywanych *symulacjami sterowanymi zdarzeniami* (ang. *event-driven simulations*), w których jedne akcje („zdarzenia”) wyzwalają dalsze zdarzenia zachodzące z pewnym opóźnieniem, a te z kolei wyzwalają dalsze zdarzenia itd.

Nasz model obliczeniowy układu będzie zawierał obiekty odpowiadające podstawowym elementom składowym, z których złożony jest układ. Mamy więc *przewody* (ang. *wires*) przenoszące *sygnały cyfrowe* (ang. *digital signals*). Sygnał cyfrowy może w każdym momencie przyjmować tylko jedną z dwóch możliwych wartości: 0 lub 1. Mamy również różnego rodzaju cyfrowe *bloki funkcjonalne* (ang. *function boxes*), które łączą przewody przenoszące sygnały wejściowe z przewodami wyjściowymi. Takie bloki wytwarzają sygnały wyjściowe obliczane na podstawie ich sygnałów wejściowych. Sygnał wyjściowy jest generowany z pewnym opóźnieniem, zależnym od rodzaju bloku funkcjonalnego. Na przykład *inverter* (ang. *inverter*; inaczej: bramka NOT) to podstawowy blok funkcjonalny odwracający (negujący) swoje wejście. Jeśli sygnał wejściowy zmienia się na 0, to z opóźnieniem charakterystycznym dla inwertera sygnał wyjściowy zmienia się na 1. Podobnie, jeśli sygnał wejściowy zmienia się na 1, to z takim samym opóźnieniem sygnał wyjściowy zmienia się na 0. Symbolicznie przedstawiamy inwerter tak, jak to widać na rys. 3.24. Bramka AND (ang. *and-gate*), również pokazana na rys. 3.24, to podstawowy blok funkcjonalny o dwóch wejściach i jednym wyjściu. Jej sygnał wyjściowy



Rys. 3.24. Podstawowe bloki funkcjonalne (bramki elementarne) w symulatorze obwodów logicznych



Rys. 3.25. Schemat półsumatora

wy jest obliczany jako *iloczyn logiczny* (ang. *logical and*) jej wejść. Oznacza to, że jeśli oba sygnały wejściowe stają się równe 1, to z charakterystycznym dla siebie opóźnieniem bramka AND ustawa swój sygnał wyjściowy na 1; w przeciwnym razie jest on równy 0. *Bramka OR* (ang. *or-gate*) jest podobnym podstawowym blokiem funkcjonalnym o dwóch wejściach, która oblicza swój sygnał wyjściowy jako *sumę logiczną* (ang. *logical or*) swoich wejść. Oznacza to, że jej wyjście jest ustawiane na 1, jeśli przynajmniej jeden z sygnałów wejściowych jest równy 1; w przeciwnym razie wyjście staje się równe 0.

Podstawowe bloki funkcjonalne możemy łączyć ze sobą, tworząc bloki realizujące bardziej złożone funkcje. Aby to osiągnąć, łączymy wyjścia jednych bloków funkcjonalnych z wejściami innych bloków. Na przykład *półsumator* (ang. *half-adder*) pokazany na rys. 3.25 składa się z jednej bramki OR, dwóch bramek AND oraz jednego inwertera. Wchodzą do niego dwa sygnały wejściowe A i B, a wychodzą dwa sygnały wyjściowe S i C. Sygnał wyjściowy S staje się równy 1, gdy dokładnie jeden z sygnałów A i B jest równy 1, a sygnał wyjściowy C staje się równy 1, gdy obydwa sygnały, A i B, są równe 1. Jak możemy zobaczyć z rysunku, ze względu na opóźnienia bramek sygnały wyjściowe mogą być generowane z różnymi opóźnieniami. Powoduje to wiele trudności przy konstruowaniu układów cyfrowych.

Zbudujemy teraz program modelujący cyfrowe układy logiczne, które chcemy badać. Program będzie tworzył obiekty obliczeniowe modelujące przewody „przechowujące” sygnały. Bloki funkcjonalne będą modelowane jako procedury wymuszające zachodzenie odpowiednich związków między sygnałami.

Jednym z podstawowych elementów naszej symulacji będzie procedura `make-wire` tworząca przewody. Możemy na przykład w następujący sposób utworzyć sześć przewodów:

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

Blok funkcjonalny łączymy z przewodami, wywołując procedurę tworzącą dany rodzaj bloku. Argumentami takiego konstruktora są przewody, które mają być podłączone do bloku. Zakładając na przykład, że potrafimy tworzyć bramki AND, bramki OR i inwertery, możemy połączyć je w półsumator pokazany na rys. 3.25:

```
(or-gate a b d)
ok

(and-gate a b c)
ok

(inverter c e)
ok

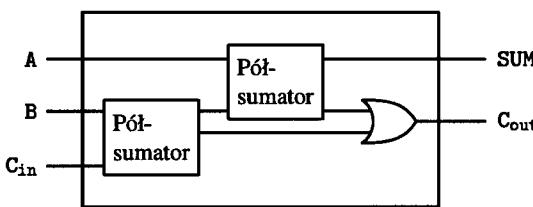
(and-gate d e s)
ok
```

Jeszcze lepiej, możemy wprost nazwać tę operację, definiując procedurę `half-adder` konstruującą półsumator na podstawie czterech zewnętrznych przewodów, które należy podłączyć do takiego układu:

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

Korzyść wynikającą z takiej definicji polega na tym, że możemy używać samej procedury `half-adder` jako elementu składowego do tworzenia bardziej złożonych układów. Na rysunku 3.26 jest na przykład pokazany *pełny sumator* (ang. *full-adder*) złożony z dwóch półsumatorów i bramki OR<sup>26</sup>. Pełny sumator możemy skonstruować w następujący sposób:

<sup>26</sup> Pełny sumator to podstawowy element używany w układach sumujących liczby binarne. Tutaj A i B to odpowiadające sobie bity dwóch sumowanych liczb, a  $C_{in}$  to bit przeniesienia pozycji o jeden na prawo. Układ ten generuje sygnał SUM, który jest odpowiednim bitem sumy, oraz sygnał  $C_{out}$ , który jest bitem przeniesienia podlegającym propagacji w lewo.



Rys. 3.26. Schemat pełnego sumatora

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

Mając tak zdefiniowaną procedurę **full-adder**, możemy jej używać jako elementu składowego do budowy jeszcze bardziej złożonych układów. (Przykład tego można znaleźć w ćwiczeniu 3.30).

W gruncie rzeczy nasz symulator dostarcza nam narzędzi do tworzenia języka układów. Jeśli przyjmiemy ogólne spojrzenie na języki, z jakim podchodziłyśmy do badania Lispu w podrozdziale 1.1, to możemy powiedzieć, że podstawowe bloki funkcjonalne tworzą pierwotne elementy języka i że łączenie bloków przewodami stanowi metodę łączenia, a określanie schematów połączeń w postaci procedur służy za sposób abstrakcji.

### Podstawowe bloki funkcjonalne

Podstawowe bloki funkcjonalne (bramki elementarne) implementują „siły”, na skutek których zmiany sygnałów w jednych przewodach powodują zmiany sygnałów w innych przewodach. Do budowy bloków funkcjonalnych używamy następujących operacji na przewodach:

- **(get-signal <przewód>)**  
daje w wyniku bieżącą wartość sygnału w przewodzie;
- **(set-signal! <przewód> <nowa wartość>)**  
zmienia wartość sygnału w przewodzie na nową wartość;
- **(add-action! <przewód> <procedura bezargumentowa>)**  
zaznacza, że dana procedura powinna być uruchamiana za każdym razem, gdy zmienia się wartość sygnału w przewodzie; takie procedury stanowią narzędzie, za pomocą którego zmiany wartości sygnału w przewodzie są przekazywane do innych przewodów.

Dodatkowo będziemy używać procedury `after-delay`, której argumentami są opóźnienie i procedura do uruchomienia, która wywołuje daną procedurę z podanym opóźnieniem.

Używając tych procedur, możemy zdefiniować podstawowe funkcje realizowane przez elementarne bramki logiczne. W celu połączenia wejścia z wyjściem za pośrednictwem inwertera używamy `add-action!`, aby związać z przewodem wejściowym procedurę, która będzie uruchamiana za każdym razem, gdy sygnał w przewodzie wejściowym zmieni wartość. Procedura ta oblicza negację logiczną (`logical-not`) sygnału wejściowego, a następnie z opóźnieniem `inverter-delay` ustawia sygnał wyjściowy na tę nową wartość:

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
        (lambda ()
          (set-signal! output new-value)))))

  (add-action! input invert-input)
  'ok)

(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Niepoprawny sygnał" s))))
```

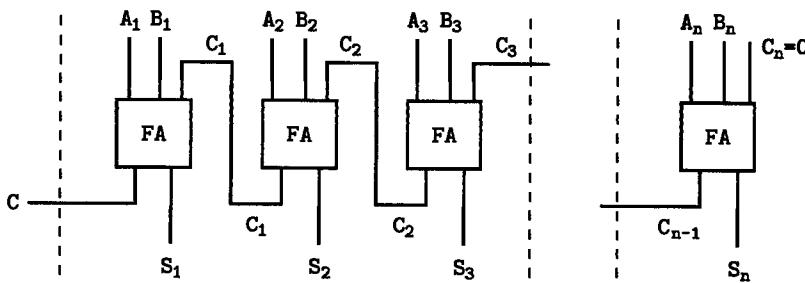
Bramka AND jest trochę bardziej skomplikowana. Procedura określająca jej działanie musi być uruchamiana, gdy którykolwiek z sygnałów wejściowych zmieni wartość. Oblicza ona iloczyn logiczny (`logical-and`) (za pomocą procedury analogicznej do `logical-not`) wartości sygnałów w przewodach wejściowych i powoduje, że nowa wartość sygnału pojawia się z opóźnieniem `and-gate-delay` w przewodzie wyjściowym.

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
          (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
        (lambda ()
          (set-signal! output new-value)))))

  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok)
```

### Ćwiczenie 3.28

Zdefiniuj bramkę OR jako podstawowy blok funkcjonalny. Twój konstruktor `or-gate` powinien być podobny do `and-gate`.



Rys. 3.27. Sumator kaskadowy dla liczb  $n$ -bitowych

### Ćwiczenie 3.29

Inny sposób otrzymania bramki OR polega na skonstruowaniu złożonego cyfrowego układu logicznego, zbudowanego z bramek AND oraz inwerterów. Zdefiniuj procedurę `or-gate`, która to realizuje. Jakie jest opóźnienie bramki OR w zależności od `and-gate-delay` i `inverter-delay`?

### Ćwiczenie 3.30

Na rysunku 3.27 widać *sumator kaskadowy* (ang. *ripple-carry adder*) powstały jako łańcuch  $n$  połączonych pełnych sumatorów (FA). Jest to najprostsza postać sumatora równoległego dodającego dwie  $n$ -bitowe liczby binarne. Sygnały wejściowe  $A_1, A_2, A_3, \dots, A_n$  i  $B_1, B_2, B_3, \dots, B_n$  stanowią dwie dodawane liczby binarne (każde  $A_k$  i  $B_k$  jest równe 0 lub 1). Układ ten generuje sygnały  $S_1, S_2, S_3, \dots, S_n$ ,  $n$  bitów sumy oraz  $C$  — przeniesienie wynikające z dodawania. Napisz procedurę `ripple-carry-adder` generującą taki układ. Argumentami tej procedury powinny być trzy listy, każda złożona z  $n$  przewodów —  $A_k, B_k$  i  $S_k$  — a także przewód  $C$ . Główną wadą sumatora kaskadowego jest konieczność czekania na propagację sygnałów przeniesienia. Jakie opóźnienie, wyrażone za pomocą opóźnień bramek AND, OR i inwerterów, jest potrzebne do wygenerowania całkowitego sygnału wyjściowego z  $n$ -bitowego sumatora kaskadowego?

### Reprezentowanie przewodów

Przewód w naszej symulacji będzie obiektem obliczeniowym z dwiema lokalnymi zmiennymi stanu: `signal-value` (wartością sygnału, początkowo równą 0) oraz `action-procedures` (kolekcją procedur-akcji, które należy wywołać, gdy wartość sygnału ulega zmianie). Przewód implementujemy, używając techniki przekazywania komunikatów, jako zestaw procedur lokalnych wraz z procedurą `dispatch`, która wybiera odpowiednią operację lokalną — tak jak to zrobiliśmy dla obiektów reprezentujących proste konta bankowe w punkcie 3.1.1:

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '()))
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))

```

```

(begin (set! signal-value new-value)
       (call-each action-procedures))
  'done))

(define (accept-action-procedure! proc)
  (set! action-procedures (cons proc action-procedures))
  (proc))

(define (dispatch m)
  (cond ((eq? m 'get-signal) signal-value)
        ((eq? m 'set-signal!) set-my-signal!)
        ((eq? m 'add-action!) accept-action-procedure!)
        (else (error "Nieznana operacja -- WIRE" m))))
  dispatch))

```

Procedura lokalna `set-my-signal!` sprawdza, czy nowa wartość sygnału różni się od dotychczasowej. Jeśli tak, to za pomocą następującej procedury `call-each`, która wywołuje procedury bezparametryczne z zadanej listy, są uruchamiane wszystkie procedury-akcje:

```

(define (call-each procedures)
  (if (null? procedures)
      'done
      (begin
        ((car procedures))
        (call-each (cdr procedures))))))

```

Procedura lokalna `accept-action-procedure!` dodaje daną procedurę do listy procedur do uruchomienia, a także wywołuje tę procedurę. (Zobacz ćwiczenie 3.31).

Przy powyższej definicji procedury `dispatch` możemy udostępnić lokalne operacje na przewodach za pomocą następujących procedur<sup>27</sup>:

```

(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))

(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))

```

<sup>27</sup> Procedury te są jedynie lukrem syntaktycznym udostępniającym zwykłą proceduralną składnię wywoływania lokalnych procedur obiektu. Zadziwiający jest fakt, że możemy tak prosto zamiieniać rolami „procedury” i „dane”. Jeśli na przykład napiszemy `(wire 'get-signal)`, to myślimy o `wire` jak o procedurze wywoływanej z komunikatem `get-signal` jako argumentem. Alternatywnie, pisząc `(get-signal wire)`, zachęcamy do myślenia o `wire` jak o obiekcie danych, który jest argumentem procedury `get-signal`. Prawda jest taka, że w języku, w którym możemy posługiwać się procedurami jak obiektami, nie ma zasadniczej różnicy między „procedurami” a „danimi” i możemy wybrać lukier syntaktyczny umożliwiający nam programowanie w dowolnym stylu, jaki wybierzemy.

Przewody, które przenoszą zmienne w czasie sygnały i mogą być stopniowo połączane do kolejnych urządzeń, są typowym przykładem obiektów modyfikowalnych. Zamodelowaliśmy je jako procedury z lokalnymi zmiennymi stanu modyfikowanymi za pomocą przypisania. Gdy tworzymy nowy przewód, tworzone są (przez wyrażenie `let` w `make-wire`) nowe zmienne stanu oraz nowa procedura `dispatch`, która jest wynikiem i która wyraża środowisko zawierające nowe zmienne stanu.

Przewody są współdzielone przez różne urządzenia, które zostały do nich podłączone. Tak więc zmiana spowodowana interakcją z jednym urządzeniem wpływa na wszystkie inne urządzenia podłączone do przewodu. Przewód komunikuje się ze swoimi sąsiadami, wywołując procedury-akcje podawane w momencie podłączania urządzeń.

## Kolejka zdarzeń

Jedyną rzeczą, jakiej brakuje do ukończenia symulatora, jest procedura *after-delay*. Pomyśl polega na utrzymywaniu struktury danych, nazywanej *kolejką zdarzeń* (ang. *agenda*), zawierającej listę rzeczy do zrobienia. Na kolejce zdarzeń są zdefiniowane następujące operacje:

- (**make-agenda**)  
daje w wyniku nową, pustą kolejkę zdarzeń;
  - (**empty-agenda?** *<kolejka zdarzeń>*)  
daje w wyniku prawdę, jeśli określona kolejka zdarzeń jest pusta;
  - (**first-agenda-item** *<kolejka zdarzeń>*)  
daje w wyniku pierwszy element kolejki zdarzeń;
  - (**remove-first-agenda-item!** *<kolejka zdarzeń>*)  
modyfikuje kolejkę zdarzeń, usuwając z niej pierwszy element;
  - (**add-to-agenda!** *<czas> <akcja> <kolejka zdarzeń>*)  
modyfikuje kolejkę zdarzeń, wstawiając do niej daną procedurę-akcję do wykonania w określonym momencie;
  - (**current-time** *<kolejka zdarzeń>*)  
daje w wyniku bieżący czas symulacji.

Konkretna kolejka zdarzeń, której używamy, jest oznaczona przez `the-agenda`. Procedura `after-delay` wstawia do niej nowe elementy:

Symulacją steruje procedura `propagate`, która operuje na kolejce zdarzeń `the-agenda`, wykonując kolejne procedury z kolejki. Ogólnie mówiąc, w momencie przebiegu symulacji nowe elementy będą wstawiane do kolejki zdarzeń, a `propagate` będzie prowadzić symulację tak długo, jak długo w kolejce zdarzeń będą jakieś elementy:

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

### Przykładowa symulacja

Następująca procedura, która umieszcza na przewodzie „próbnik”, ukazuje nam działanie symulatora. Próbnik każe przewodowi, żeby za każdym razem gdy wartość jego sygnału się zmieni, wypisał nową wartość sygnału razem z bieżącym czasem i nazwą przewodu:

```
(define (probe name wire)
  (add-action! wire
    (lambda ()
      (newline)
      (display name)
      (display " ")
      (display (current-time the-agenda))
      (display " Nowa wartość = ")
      (display (get-signal wire)))))
```

Zaczynamy od zainicjowania kolejki zdarzeń i określenia opóźnień podstawowych bloków funkcjonalnych:

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

Potem definiujemy cztery przewody, umieszczając próbniki na dwóch z nich:

```
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))

(probe 'sum sum)
sum 0 Nowa wartość = 0
```

```
(probe 'carry carry)
carry 0 Nowa wartość = 0
```

Następnie łączymy przewody w półsumator (jak było to pokazane na rys. 3.25), ustawiamy sygnał w przewodzie `input-1` na 1 i uruchamiamy symulację:

```
(half-adder input-1 input-2 sum carry)
```

```
ok
```

```
(set-signal! input-1 1)
```

```
done
```

```
(propagate)
```

```
sum 8 Nowa wartość = 1
```

```
done
```

Sygnał `sum` zmienia się na 1 w chwili 8. Minęło więc 8 jednostek czasu od początku symulacji. W tym momencie możemy ustawić sygnał w przewodzie `input-2` na 1 i pozwolić na propagację sygnałów:

```
(set-signal! input-2 1)
```

```
done
```

```
(propagate)
```

```
carry 11 Nowa wartość = 1
```

```
sum 16 Nowa wartość = 0
```

```
done
```

Sygnał `carry` zmienia się na 1 w chwili 11, a sygnał `sum` zmienia się na 0 w chwili 16.

### Ćwiczenie 3.31

Procedura wewnętrzna `accept-action-procedure!` zdefiniowana w `make-wire` określa, że gdy dodajemy nową procedurę-akcję do przewodu, wtedy jest ona natychmiast uruchamiana. Wyjaśnij, dlaczego takie inicjowanie jest konieczne. W szczególności prześledź we wcześniejszych akapitach przykład półsumatora i powiedz, czym różniłaby się odpowiedź systemu, gdybyśmy zdefiniowali `accept-action-procedure!` w następujący sposób:

```
(define (accept-action-procedure! proc)
  (set! action-procedures (cons proc action-procedures)))
```

### Implementowanie kolejki zdarzeń

Na koniec podajemy szczegóły struktury danych kolejki zdarzeń zawierającej procedury zaplanowane do wykonania w przyszłości.

Kolejka zdarzeń składa się z przedziałów czasu (ang. *time segments*). Każdy przedział czasu to para składająca się z liczby (czasu) i kolejki (zob. ćwicze-

nie 3.32) zawierającej procedury, które należy wykonać w danym przedziale czasu.

```
(define (make-time-segment time queue)
  (cons time queue))

(define (segment-time s) (car s))

(define (segment-queue s) (cdr s))
```

Będziemy operować na kolejkach zawartych w przedziałach czasu, korzystając z operacji na kolejkach opisanych w punkcie 3.3.2.

Kolejka zdarzeń sama jest jednowymiarową tablicą przedziałów czasu. Różni się ona od tablic opisanych w punkcie 3.3.3 tym, że przedziały są w niej pamiętane w kolejności rosnącego czasu. Dodatkowo w atrapie kolejki zdarzeń pamiętamy *bieżący czas* (tzn. czas wykonania ostatniej akcji). Nowo utworzona kolejka zdarzeń nie zawiera żadnych przedziałów czasu, a jej bieżący czas jest równy 0<sup>28</sup>:

```
(define (make-agenda) (list 0))

(define (current-time agenda) (car agenda))

(define (set-current-time! agenda time)
  (set-car! agenda time))

(define (segments agenda) (cdr agenda))

(define (set-segments! agenda segments)
  (set-cdr! agenda segments))

(define (first-segment agenda) (car (segments agenda)))

(define (rest-segments agenda) (cdr (segments agenda)))
```

Kolejka zdarzeń jest pusta, jeśli nie zawiera żadnych przedziałów czasu:

```
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

Chcąc dodać do kolejki zdarzeń akcję, najpierw sprawdzamy, czy kolejka zdarzeń jest pusta. Jeśli tak, to tworzymy przedział czasu i umieszczamy go w kolejce zdarzeń. W przeciwnym razie przeglądamy kolejkę zdarzeń, sprawdzając czas każdego przedziału. Jeśli znajdziemy przedział o wyznaczonym czasie, dodajemy akcję do związanego z nim kolejki. Jeśli dojdziemy do czasu

<sup>28</sup> Kolejka zdarzeń, tak jak tablice opisane w punkcie 3.3.3, jest listą z atrapą, jednak ze względu na czas pamiętany w atrapie nie potrzeba żadnej atrapy elementu (takiej jak symbol `*table*` używany w tablicach).

poźniejszego od wyznaczonego, to tuż przed nim wstawiamy do kolejki zdarzeń nowy przedział czasu. Jeśli dojdziemy do końca kolejki zdarzeń, musimy na jej końcu dodać nowy przedział czasu.

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments))))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments))
                      action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr!
               segments
               (cons (make-new-time-segment time action)
                     (cdr segments)))
              (add-to-segments! rest))))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments!
         agenda
         (cons (make-new-time-segment time action)
               segments))
        (add-to-segments! segments))))
```

Procedura usuwająca pierwszy element z kolejki zdarzeń usuwa element z początku kolejki pierwszego przedziału czasu. Jeśli na skutek tego przedział czasu staje się pusty, usuwamy go z listy przedziałów<sup>29</sup>:

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

<sup>29</sup> Zwróćmy uwagę, że wyrażenie `if` w tej procedurze nie ma *(alternatywy)*. Takie „jednoczłonowe instrukcje `if`” są używane raczej do rozstrzygania, czy coś wykonać, a nie do wybierania jednego z dwóch podwyrażeń. Wartość wyrażenia `if` jest nieokreślona, jeżeli predykat jest fałszywy i nie ma *(alternatywy)*.

Pierwszy element znajdujący się na początku kolejki zdarzeń to element znajdujący się na początku pierwszego przedziału czasu. Za każdym razem, gdy wydobywamy element, aktualniamy również bieżący czas<sup>30</sup>:

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Pusta kolejka zdarzeń -- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

### Ćwiczenie 3.32

Procedury, które należy uruchomić w każdym przedziale czasu kolejki zdarzeń, są pamiętane w kolejce. Tak więc w każdym przedziale czasu procedury są wywoływane w kolejności, w jakiej były wstawiane do kolejki zdarzeń (kolejność FIFO). Wyjaśnij, dlaczego należy stosować taką właśnie kolejność. W szczególności prześledź zachowanie bramki AND, której wejścia zmieniają swoje wartości z 0 i 1 na 1 i 0 w tym samym przedziale czasu, i powiedz, jak zmieniłoby się jej zachowanie, gdybyśmy pamiętali procedury przedziału czasu na zwykłą liście, dodając i usuwając procedury tylko z początku listy (tzw. kolejność LIFO; ang. *last in, first out* — ostatni na wejściu, pierwszy na wyjściu).

### 3.3.5. Propagacja więzów

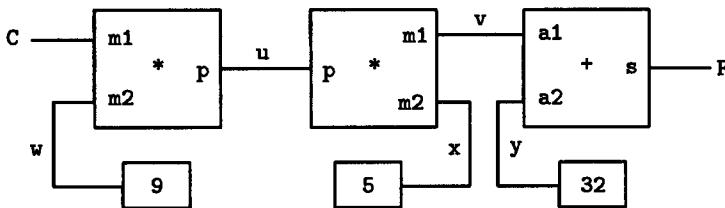
Programy komputerowe mają tradycyjnie jednokierunkową strukturę obliczeń — obliczają potrzebne wyniki, wykonując operacje na określonych wcześniej danych wejściowych. Jednakże często modelujemy systemy za pomocą związków między różnymi wielkościami. Na przykład matematyczny model układu mechanicznego może zawierać informację, że odkształcenie  $d$  metalowego pręta jest związane z siłą  $F$  przyłożoną do pręta, jego długością  $L$ , polem jego przekroju poprzecznego  $A$  oraz modułem sprężystości  $E$  za pomocą następującego wzoru:

$$dAE = FL$$

Takie równanie nie jest jednokierunkowe. Mając dane dowolne cztery wielkości, możemy go użyć do obliczenia piątej. Mimo to zapisanie tego równania w tradycyjnym języku programowania wymusiłoby na nas wybranie jednej wielkości, która ma być obliczona na podstawie pozostałych czterech. Tak więc nie dałoby się zastosować procedury obliczającej pole  $A$  do obliczenia odkształcenia  $d$ , mimo że obliczenie  $A$  i  $d$  opiera się na tym samym wzorze<sup>31</sup>.

<sup>30</sup> W ten sposób bieżący czas będzie zawsze czasem ostatnio wykonanej akcji. Przechowywanie tego czasu w atrapie kolejki zdarzeń umożliwia nam dostęp do niego, nawet jeśli związany z nim przedział czasu zostanie usunięty.

<sup>31</sup> Propagacja więzów pojawiła się po raz pierwszy w niezwykle perspektywicznym systemie SKETCHPAD opisany przez Ivana Sutherlanda w [102]. Piękny system propagacji więzów,



Rys. 3.28. Związek  $9C = 5(F - 32)$  wyrażony jako sieć więzów

W niniejszym punkcie naszkicujemy konstrukcję języka umożliwiającego posługiwanie się związkami jako takimi. Pierwotnymi elementami tego języka są więzy pierwotne (ang. *primitive constraints*), które wyrażają określone związki zachodzące między wielkościami. Na przykład „suma” (adder  $a + b = c$ ), „iloczyn” (multiplier  $x \cdot y = z$ ) wyraża więzy  $xy = z$ , a „stała” (constant 3.14  $\pi$ ) określa, że wartość  $x$  musi być równa 3.14.

Nasz język umożliwia łączenie więzów pierwotnych w celu wyrażania bardziej złożonych związków. Łączymy więzy, tworząc sieć więzów (ang. *constraint network*), w której więzy są połączone za pomocą złącz (ang. *connectors*). Złącze to obiekt „przechowujący” wartość, która może występować w jednych lub więcej więzach. Wiadomo na przykład, że związek między skalą temperatur Fahrenheita i Celsjusza wyraża się wzorem

$$9C = 5(F - 32)$$

O takich więzach możemy myśleć jak o sieci składającej się z pierwotnych więzów sumy, iloczynu i stałych (rys. 3.28). Na rysunku, po lewej stronie, widzimy pudełko przedstawiające iloczyn z trzema końcówkami oznaczonymi  $m_1$ ,  $m_2$  i  $p$ . Łączącą one iloczyn z resztą systemu w następujący sposób: Końcówka  $m_1$  jest połączona ze złączem  $C$ , które przechowuje temperaturę w stopniach Celsjusza. Końcówka  $m_2$  jest połączona ze złączem  $w$ , które jest również połączone z pudełkiem stałej, zawierającym liczbę 9. Końcówka  $p$ , która pudełko iloczynu ustala jako iloczyn  $m_1$  i  $m_2$ , jest połączona z końcówką  $p$  drugiego pudełka iloczynu, którego końcówka  $m_2$  jest połączona ze stałą 5, a końcówka  $m_1$  jest połączona z jednym ze składników sumy.

Obliczenia są wykonywane przez taką sieć w następujący sposób: Gdy złącze otrzymuje (od użytkownika lub od pudełka więzów, do którego jest połączone) wartość, budzi wszystkie więzy, do których jest podłączone (z wyjątkiem więzów, które obudziły to złącze), aby poinformować je, że otrzymało

oparty na języku Smalltalk, został opracowany przez Alana Borninga [8] w Xerox Palo Alto Research Center. Sussman, Stallman i Steele zastosowali propagację więzów do analizy obwodów elektronicznych [98, 99]. TK!Solver [61] stanowi obszerne środowisko modelowania oparte na wykorzystaniu więzów.

wartość. Każde obudzone pudełko więzów przegląda wówczas swoje złącza w celu sprawdzenia, czy ma dosyć informacji, żeby ustalić wartość któregoś ze złączy. Jeśli tak, to ustawia wartość tego złącza, które z kolei budzi wszystkie związane z nim więzy itd. Na przykład w przekształceniu między skalą Celsjusza i Fahrenheita złącza  $w$ ,  $x$  i  $y$  są natychmiast ustawiane za pomocą pudełek stałych na wartości, odpowiednio, 9, 5 i 32. Złącza budzą iloczyny i sumę, ale te stwierdzają, że nie mają dostatecznej ilości informacji, żeby działać. Jeśli jednak użytkownik (lub jakąś inną część sieci) określi wartość  $C$  (powiedzmy na 25), to lewy iloczyn zostanie obudzony i ustali wartość  $u$  na  $25 \cdot 9 = 225$ . Następnie  $u$  obudzi drugi iloczyn, który ustawia  $v$  na 45 i obudzi sumę, która ustawi  $F$  na 77.

### Zastosowanie systemu więzów

Aby zastosować system więzów do wykonania nakreślonego powyżej przekształcenia temperatur, musimy najpierw utworzyć dwa złącza C i F, wywołując konstruktor `make-connector`, i połączyć je odpowiednią siecią:

```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F)
ok
```

Procedura, która tworzy sieć, jest zdefiniowana następująco:

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

Procedura ta tworzy wewnętrzne złącza  $u$ ,  $v$ ,  $w$ ,  $x$  i  $y$  oraz łączy je w sposób pokazany na rys. 3.28 za pomocą konstruktorów więzów pierwotnych `adder`, `multiplier` i `constant`. Tak jak w przypadku symulatora układów cyfrowych z punktu 3.3.4, wyrażanie takich połączeń elementów pierwotnych w postaci procedur automatycznie wprowadza do naszego języka środki abstrakcji obiektów złożonych.

Aby zobaczyć sieć w działaniu, możemy umieścić na złączach C i F próbniki, używając procedury `probe` podobnej do tej, której użyliśmy w punkcie 3.3.4 do monitorowania przewodów. Umieszczenie na złączu próbnika spowoduje wypisanie komunikatu za każdym razem, gdy złącze otrzyma wartość:

```
(probe "Temperatura w skali Celsjusza" C)
(probe "Temperatura w skali Fahrenheita" F)
```

Następnie ustawiamy wartość C na 25. (Trzeci argument `set-value!` mówi C, że polecenie to pochodzi od użytkownika (ang. *user*)).

```
(set-value! C 25 'user)
Probe: Temperatura w skali Celsjusza = 25
Probe: Temperatura w skali Fahrenheita = 77
done
```

Próbnik na złączu C budzi się i podaje wartość. Złącze C rozsyła, w opisany powyżej sposób, swoją wartość po sieci. W rezultacie F otrzymuje wartość 77, co jest sygnalizowane przez próbnik podłączony do F.

Możemy teraz spróbować ustawić F na nową wartość, powiedzmy 212:

```
(set-value! F 212 'user)
Error! Sprzeczność (77 212)
```

Złącze sygnalizuje, że wykryło sprzeczność — jego wartość wynosi 77, a ktoś próbuje ustawić ją na 212. Jeżeli naprawdę chcemy ponownie użyć sieci z innymi wartościami, możemy kazać C zapomnieć swoją starą wartość:

```
(forget-value! C 'user)
Probe: Temperatura w skali Celsjusza = ?
Probe: Temperatura w skali Fahrenheita = ?
done
```

Złącze C odkrywa, że użytkownik (*user*), który początkowo podał jego wartość, wycofuje się teraz z tego, więc C zgadza się na utratę swojej wartości, co widać dzięki próbnikowi, i informuje o tym resztę sieci. Informacja ta dociera w końcu do złącza F, które stwierdza, że nie ma dalszych podstaw wierzyć, że jego wartość jest równa 77. Tak więc F również rezygnuje ze swojej wartości, co jest sygnalizowane przez próbnik.

Teraz, gdy F nie ma już żadnej wartości, możemy ustawić ją na 212:

```
(set-value! F 212 'user)
Probe: Temperatura w skali Fahrenheita = 212
Probe: Temperatura w skali Celsjusza = 100
done
```

Ta nowa wartość, gdy już rozejdzie się po sieci, zmusza C do przyjęcia wartości 100, co jest odnotowywane przez próbnik podłączony do tego złącza. Zauważmy, że dokładnie ta sama sieć została użyta do obliczenia C na podstawie F, jak i do obliczenia F na podstawie C. Taka bezkierunkowość obliczeń jest wyróżniającą cechą systemów opartych na więzach.

### Implementowanie systemu więzów

System więzów jest zaimplementowany przy użyciu obiektów proceduralnych ze stanami lokalnymi, w bardzo podobny sposób do symulatora układów cyfrowych z punktu 3.3.4. Mimo że pierwotne obiekty systemu więzów są trochę bardziej złożone, cały system jest prostszy, gdyż nie musimy utrzymywać kolejki zdarzeń ani przejmować się opóźnieniami.

Na złączach mamy określone następujące operacje podstawowe:

- **(has-value? <złącze>)**  
sprawdza, czy złącze ma określoną wartość;
- **(get-value <złącze>)**  
daje w wyniku bieżącą wartość złącza;
- **(set-value! <złącze> <nowa wartość> <informator>)**  
wskazuje, że informator prosi złącze o ustawienie swojej wartości na nową wartość;
- **(forget-value! <złącze> <wycofując>)**  
informuje złącze, że wycofując prosi o zapomnienie wartości;
- **(connect <złącze> <nowe więzy>)**  
informuje złącze o podłączeniu do nowych więzów.

Złącza komunikują się z więzami za pomocą procedur: **inform-about-value**, która informuje więzy o tym, że złącze otrzymało wartość, oraz **inform-about-no-value**, która informuje więzy o tym, że złącze straciło swoją wartość.

Procedura **adder** tworzy więzy między sumowanymi złączami a1 i a2 a ich sumą reprezentowaną przez złącze sum. Więzy sumy są zaimplementowane w postaci procedury ze stanem lokalnym (procedura **me** poniżej):

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2))
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
```

```

        (- (get-value sum) (get-value a1))
        me))
((and (has-value? a2) (has-value? sum))
 (set-value! a1
        (- (get-value sum) (get-value a2))
        me))))
(define (process-forget-value)
  (forget-value! sum me)
  (forget-value! a1 me)
  (forget-value! a2 me)
  (process-new-value))
(define (me request)
  (cond ((eq? request 'I-have-a-value)
         (process-new-value))
        ((eq? request 'I-lost-my-value)
         (process-forget-value))
        (else
         (error "Nieznana operacja -- ADDER" request))))
(connect a1 me)
(connect a2 me)
(connect sum me)
me)

```

Procedura adder podłącza określone złącza do nowych więzów sumy, które są również jej wynikiem. Procedura me, która reprezentuje więzy, działa jak procedura rozdzielająca sterowanie między procedury lokalne. Przy rozdzielaniu tym jest używany następujący „interfejs składniowy” (zob. przypis 27 w punkcie 3.3.4):

```

(define (inform-about-value constraint)
  (constraint 'I-have-a-value))

(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))

```

Lokalna procedura więzów sumy process-new-value jest wywoływaną, gdy więzy są informowane o tym, że jedno ze złączy otrzymało wartość. Więzy najpierw sprawdzają, czy obydwa składniki a1 i a2 mają określone wartości. Jeśli tak, to przekazują złączu sum informację, aby ustawiło swoją wartość na sumę wartości składników. Argument informant procedury set-value! jest równy me, co reprezentuje sam obiekt więzów. Jeśli a1 lub a2 nie ma określonej wartości, więzy sumy sprawdzają, czy może a1 i sum mają określone wartości. Jeśli tak, wartość a2 jest ustawiana na różnicę tych dwóch wartości. Wreszcie, jeśli a2 i sum mają określone wartości, pozwala to więzom sumy określić a1. Jeśli więzy sumy dowiadują się, że jedno ze złączy straciło swoją wartość, wszystkie złącza są informowane, że straciły swoje wartości. (Jednak tylko

te wartości, które były ustawione przez te więzy, są faktycznie niszczone). Następnie jest uruchamiana procedura `process-new-value`. Dzieje się tak, gdyż jedno lub więcej złączy może nadal mieć określone wartości (tzn. złącza mogą mieć wartości, które nie były początkowo ustawiane przez te więzy) i te wartości należy rozesłać z powrotem przez więzy sumy.

Więzy iloczynu są bardzo podobne do więzów sumy. Jeśli jeden z czynników jest równy 0, to iloczyn (product) jest ustawiany na 0, nawet jeżeli drugi z czynników nie jest znany.

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                         (* (get-value m1) (get-value m2))
                         me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2
                         (/ (get-value product) (get-value m1))
                         me))
          ((and (has-value? product) (has-value? m2))
           (set-value! m1
                         (/ (get-value product) (get-value m2))
                         me))))
    (define (process-forget-value)
      (forget-value! product me)
      (forget-value! m1 me)
      (forget-value! m2 me)
      (process-new-value))
    (define (me request)
      (cond ((eq? request 'I-have-a-value)
             (process-new-value))
            ((eq? request 'I-lost-my-value)
             (process-forget-value))
            (else
              (error "Nieznana operacja -- MULTIPLIER" request))))
    (connect m1 me)
    (connect m2 me)
    (connect product me)
    me)
```

Konstruktor `constant` ustawia po prostu wartość określonego złącza. Każdy komunikat `I-have-a-value` lub `I-lost-my-value` wysłany do pudełka więzów stałych spowoduje błąd.

---

```
(define (constant value connector)
  (define (me request)
    (error "Nieznana operacja -- CONSTANT" request))
  (connect connector me)
  (set-value! connector value me)
  me)
```

Wreszcie, próbnik wypisuje komunikat o ustawieniu lub wymazaniu wartości określonego złącza:

```
(define (probe name connector)
  (define (print-probe value)
    (newline)
    (display "Probe: ")
    (display name)
    (display " = ")
    (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value)
    (print-probe "?"))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error "Nieznana operacja -- PROBE" request))))
  (connect connector me)
  me)
```

### Reprezentowanie złączy

Złącze jest reprezentowane w postaci obiektu proceduralnego z lokalnymi zmiennymi stanu: `value` — bieżąca wartość złącza; `informant` — obiekt, który ustawił wartość złącza; `constraints` — lista więzów, do których złącze jest podłączone.

```
(define (make-connector)
  (let ((value false) (informant false) (constraints '()))
    (define (set-my-value newval setter)
      (cond ((not (has-value? me))
             (set! value newval)
             (set! informant setter)
             (for-each-except setter
                               inform-about-value
                               constraints))
            ((not (= value newval))
             (error "Nieznana operacja -- SET-MY-VALUE" value newval)))
```

```

(error "Sprzeczność" (list value newval)))
(else 'ignored)))
(define (forget-my-value retractor)
  (if (eq? retractor informant)
      (begin (set! informant false)
             (for-each-except retractor
               inform-about-no-value
               constraints)))
      'ignored))
(define (connect new-constraint)
  (if (not (memq new-constraint constraints))
      (set! constraints
            (cons new-constraint constraints)))
  (if (has-value? me)
      (inform-about-value new-constraint)
      'done))
(define (me request)
  (cond ((eq? request 'has-value?)
         (if informant true false))
        ((eq? request 'value) value)
        ((eq? request 'set-value!) set-my-value)
        ((eq? request 'forget) forget-my-value)
        ((eq? request 'connect) connect)
        (else (error "Nieznaną operacją -- CONNECTOR"
                     request))))
  me))

```

Procedura lokalna `złącza` `set-my-value` jest wywoływaną, gdy ktoś chce ustawić wartość `złącza`. Jeśli `złącze` nie ma aktualnie żadnej wartości, to przyjmuje podaną wartość i zapamiętuje (w zmiennej `informant`), od kogo pochodzi ta wartość<sup>32</sup>. Następnie `złącze` informuje wszystkie więzy, do których jest podłączone, z wyjątkiem tych, od których otrzymało wartość, o swojej nowej wartości. Jest to wykonywane za pomocą następującego iteratora, który stosuje daną procedurę do wszystkich elementów danej listy z wyjątkiem jednego danego elementu:

```

(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items))))))
  (loop list))

```

<sup>32</sup> Wartość ta nie musi pochodzić od `złącza`. W naszym przykładzie z temperaturami używaliśmy symbolu `user` w miejscu parametru `setter`.

Jeśli złącze zostanie poproszone o wymazanie swojej wartości, to uruchamiana jest procedura lokalna `forget-my-value`, która najpierw upewnia się, czy żądanie pochodzi od tego samego obiektu, który ustawił początkowo wartość złącza. Jeśli tak, złącze informuje wszystkie więzy, do których jest połączone, o utracie wartości.

Procedura lokalna `connect` dodaje dane nowe więzy do listy więzów, o ile nie występują już one na tej liście. Następnie, jeśli złącze ma określoną wartość, informuje o tym fakcie nowe więzy.

Procedura złącza `me` służy jako procedura rozdzielająca sterowanie do pozostałych procedur wewnętrznych, a także reprezentuje złącze jako obiekt. Następujące procedury określają interfejs składniowy procedury rozdzielającej:

```
(define (has-value? connector)
  (connector 'has-value?))

(define (get-value connector)
  (connector 'value))

(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))

(define (forget-value! connector retractor)
  ((connector 'forget) retractor))

(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

### Ćwiczenie 3.33

Używając pierwotnych więzów iloczynu, sumy i stałych, zdefiniuj procedurę `averager`, której argumentami są trzy złącza `a`, `b` i `c` i która ustanawia więzy wyrażające, że wartość `c` jest średnią z wartości `a` i `b`.

### Ćwiczenie 3.34

Ludwik Myślicielak chce zbudować więzy podnoszenia do kwadratu — takie urządzenie o dwóch końcówkach, że wartość złącza `b` podłączonego do drugiej końcówki będzie zawsze kwadratem wartości złącza `a` podłączonego do pierwszej końcówki. Proponuje on następujące proste urządzenie zbudowane z więzów iloczynu:

```
(define (squarer a b)
  (multiplier a a b))
```

W takim rozwiążaniu kryje się poważny błąd. Wyjaśnij, jaki.

### Ćwiczenie 3.35

Ben Bajerbit powiedział Ludwikowi, że jeden ze sposobów uniknięcia problemu z ćwiczeniem 3.34 polega na zdefiniowaniu więzów podnoszenia do kwadratu w postaci no-

wych więzów pierwotnych. Wypełnij brakujące miejsca w wykonanym przez Bena szkicu procedury implementującej takie więzy:

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "Kwadrat mniejszy od 0 -- SQUARER" (get-value b))
            (możliwość1))
        (możliwość2)))
  (define (process-forget-value) (treść1))
  (define (me request) (treść2))
  {reszta definicji}
  me)
```

### Ćwiczenie 3.36

Przypuśćmy, że w środowisku globalnym obliczamy następujący ciąg wyrażeń:

```
(define a (make-connector))
(define b (make-connector))
(set-value! a 10 'user)
```

W pewnym momencie, w trakcie obliczania `set-value!`, jest obliczane następujące wyrażenie pochodzące z lokalnej procedury złącza:

```
(for-each-except setter inform-about-value constraints)
```

Narysuj diagram ukazujący środowisko, w którym jest obliczane powyższe wyrażenie.

### Ćwiczenie 3.37

Procedura `celsius-fahrenheit-converter` wydaje się trochę nieporęczna, gdy porównać ją z następującą definicją, zapisaną w stylu bardziej zorientowanym na wyrażenia:

```
(define (celsius-fahrenheit-converter x)
  (c+ (c* (c/ (cv 9) (cv 5))
            x)
       (cv 32)))

(define C (make-connector))
(define F (celsius-fahrenheit-converter C))
```

Operacje `c+`, `c*` itd. oznaczają tutaj „więzowe” wersje operacji arytmetycznych. Na przykład argumentami `c+` są dwa złącza, a wynikiem jest złącze połączone z nimi więzami sumy:

```
(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))
```

Zdefiniuj analogiczne procedury `c-`, `c*`, `c/` i `cv` (stała), pozwalające na takie definiowanie więzów złożonych jak w powyższym przykładzie<sup>33</sup>.

### 3.4. Współbieżność — istotny jest czas

Przekonaliśmy się o sile obiektów obliczeniowych ze stanami lokalnymi jako narzędzi do modelowania. Jednak, jak ostrzegaliśmy w punkcie 3.1.3, siła ta ma swoją cenę: utratę niewrażliwości na odniesienia, która, poprzez gąszcz wątpliwości dotyczących niezmienności i zmiany, doprowadziła nas do porzucenia podstawieniowego modelu obliczeń na rzecz bardziej zawiłego modelu środowiskowego.

Główny problem, jaki kryje się za złożonością stanów, niezmiennością i zmianą, polega na tym, że wprowadzając przypisanie, jesteśmy zmuszeni uwzględnić w naszym modelu obliczeniowym *czas*. Dopóki nie wprowadziliśmy przypisania, wszystkie nasze programy były ponadczasowe w tym sensie, że każde wyrażenie, które miało jakąś wartość, zawsze miało tę samą wartość. Jako przeciwieństwo tego przypomnijmy sobie przykład z początku punktu 3.1.1, w którym były modelowane wypłaty z konta bankowego, a wynikami były salda konta:

`(withdraw 25)`

`75`

<sup>33</sup> Zapisywanie więzów w formie wyrażeń jest wygodne, gdyż unika się konieczności nazywania wartości pośrednich w obliczeniach. Nasze oryginalne sformułowanie języka więzów jest tak samo nieporęczne jak wiele języków, gdy zajmujemy się operacjami na złożonych danych. Gdybyśmy na przykład chcieli obliczyć iloczyn  $(a + b) \cdot (c + d)$ , gdzie zmienne reprezentują wektory, moglibyśmy przyjąć „imperatywny styl”, używając procedur, które ustawiają wartości danych argumentów wektorowych, ale których wynikami nie są wektory:

```
(v-sum a b temp1)
(v-sum c d temp2)
(v-prod temp1 temp2 answer)
```

Alternatywnie, moglibyśmy, zapisując wyrażenia, używać procedur, których wynikami są wektory, i tym samym uniknąć jawnego wspomniania o `temp1` i `temp2`:

```
(define answer (v-prod (v-sum a b) (v-sum c d)))
```

Ponieważ Lisp pozwala, żeby wynikami procedur były obiekty złożone, możemy przekształcić nasz imperatywny język więzów w język w stylu zorientowanym na wyrażenia, jak jest to pokazane w niniejszym ćwiczeniu. W językach o zubożonych możliwościach obsługi obiektów złożonych, takich jak Algol, Basic czy Pascal (o ile nie posługujemy się jawnie wskaźnikami do zmiennych), gdy operujemy na obiektach złożonych, jesteśmy skazani na styl imperatywny. Można się spytać, czy mogąc korzystać z formy zapisu zorientowanej na wyrażenia, mamy jakikolwiek powód, aby implementować system w stylu imperatywnym, tak jak to zrobiliśmy w niniejszym punkcie. Jednym z powodów jest to, że język więzów niezorientowany na wyrażenia umożliwia uchwycenie zarówno obiektów więzów (np. wyników procedury `adder`), jak i obiektów złącznych. Jest to przydatne, gdy chcemy rozszerzyć system o nowe operacje, które komunikują się z więzami bezpośrednio, a nie pośrednio poprzez operacje na złączach. Chociaż zaimplementowanie stylu zorientowanego na wyrażenia za pomocą implementacji imperatywnej jest łatwe, jednak wykonanie tego w drugą stronę jest bardzo trudne.

(withdraw 25)

50

Tutaj kolejne obliczenia tego samego wyrażenia dają różne wyniki. Zachowanie takie wynika z faktu, że wykonanie instrukcji przypisania (w tym przypadku przypisań do zmiennej `balance`) określa *momenty w czasie*, w których wartości ulegają zmianie. Wynik obliczenia wyrażenia zależy nie tylko od samego wyrażenia, lecz także od tego, czy obliczenie ma miejsce przed takimi momentami, czy też po nich. Budowanie modeli przy użyciu obiektów obliczeniowych ze stanami lokalnymi zmusza nas do zmierzenia się z czasem jako zasadniczym pojęciem w programowaniu.

Możemy iść dalej w nadawaniu modelom obliczeniowym struktury odpowiadającej naszemu postrzeganiu świata fizycznego. Obiekty w rzeczywistości nie zmieniają się kolejno, po jednym w danej chwili. Postrzegamy je raczej jako działające *współbieżnie* (ang. *concurrently*) — wszystkie naraz. Zatem często naturalne jest, żeby modelować systemy w postaci kolekcji procesów obliczeniowych wykonywanych współbieżnie. Tak jak można modularyzować programy, budując modele z obiektów mających oddzielne stany lokalne, tak samo często właściwe jest, żeby podzielić model obliczeniowy na części, które działają niezależnie i współbieżnie. Nawet jeżeli programy mają być wykonywane na komputerze sekwencyjnym, zwyczaj zapisywania programów tak, jak gdyby miały być wykonywane współbieżnie, zmusza programistów do unikania nieistotnych ograniczeń czasowych, a przez to sprawia, że programy są bardziej modularne.

Oprócz tego, że programy są bardziej modularne, obliczenia współbieżne mogą być szybsze niż obliczenia sekwencyjne. Komputery sekwencyjne wykonują tylko jedną operację w danej chwili, więc czas potrzebny do wykonania zadania jest proporcjonalny do łącznej liczby wykonywanych operacji<sup>34</sup>. Jednakże, jeżeli można rozbić problem na stosunkowo niezależne fragmenty, które tylko z rzadka muszą się komunikować, to może być możliwe przydzielenie tym fragmentom oddzielnych procesorów, co daje przyspieszenie obliczeń proporcjonalne do liczby dostępnych procesorów.

Niestety, problemy związane z przypisaniem stają się jeszcze bardziej złożone, jeżeli uwzględnimy współbieżność. Współbieżne wykonywanie programów, czy to spowodowane tym, że świat działa równolegle, czy też tym, że tak działają komputery, dodatkowo komplikuje nasze rozumienie czasu.

<sup>34</sup> W rzeczywistości większość procesorów wykonuje kilka operacji naraz, stosując strategię zwaną *przetwarzaniem potokowym* (ang. *pipelining*). Mimo że ta technika znacznie polepsza efektywność wykorzystania sprzętu, jest stosowana tylko do przyspieszania wykonywania sekwencyjnego strumienia instrukcji, nie zmieniając działania programów sekwencyjnych.

### 3.4.1. Natura czasu w systemach współbieżnych

Na pozór pojęcie czasu wydaje się proste. Jest to porządek narzucony zdarzeniom<sup>35</sup>. Dla dowolnych dwóch zdarzeń *A* i *B*: albo *A* zachodzi przed *B*, albo *A* i *B* zachodzą równocześnie, albo *A* zachodzi po *B*. Sięgnijmy do przykładu konta bankowego — przyjmijmy, że Piotr wypłaca 10, a Paweł 25 dolarów z ich wspólnego konta, na którym początkowo jest 100 dolarów, a po obu wypłatach pozostaje 65 dolarów. W zależności od kolejności wypłat kolejne salda konta będą wynosić: 100 → 90 → 65 lub 100 → 75 → 65. W komputerowej implementacji systemu bankowego takie zmiany salda można zamodelować za pomocą kolejnych przypisań do zmiennej *balance*.

Jednak w skomplikowanych sytuacjach takie spojrzenie na sprawę może być problematyczne. Przypuśćmy, że Piotr i Paweł, a oprócz nich także inni ludzie korzystają z tego samego konta bankowego poprzez sieć bankomatów rozsianych po całym świecie. Faktyczny ciąg zmian salda konta będzie w istotny sposób zależał od momentów uzyskania dostępu do konta oraz szczegółów komunikacji z bankomatami.

Taka nieokreśloność kolejności zdarzeń może stwarzać poważne problemy przy konstruowaniu systemów współbieżnych. Przypuśćmy na przykład, że wypłaty realizowane przez Piotra i Pawła są zaimplementowane w postaci dwóch oddzielnych procesów współdzielących jedną zmienną *balance*, a każdy z tych procesów jest określony przez procedurę z punktu 3.1.1:

```
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Brak środków na koncie"))
```

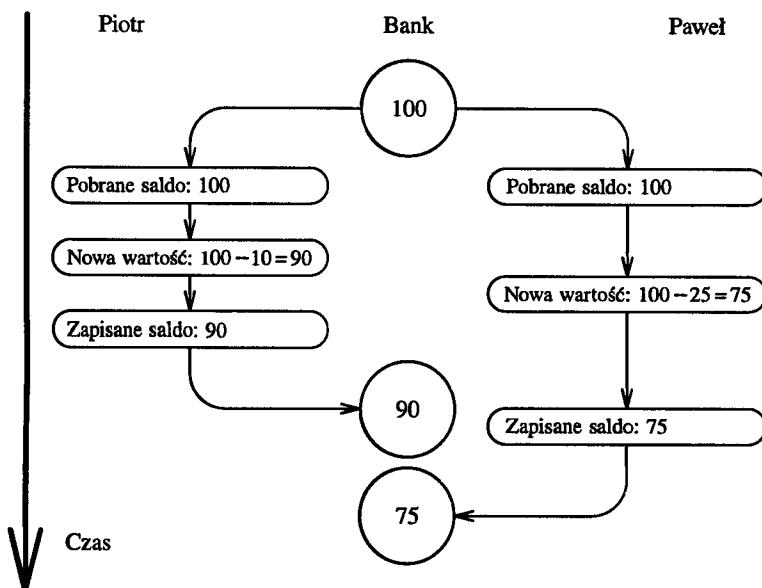
Jeśli obydwa procesy działają niezależnie, to Piotr może sprawdzać saldo konta i próbować wypłacić dopuszczalną kwotę. Jednak Paweł może wypłacić trochę pieniędzy między momentem, w którym Piotr sprawdza stan konta, a momentem, w którym wypłaca pieniądze, tym samym unieważniając sprawdzenie salda konta dokonane przez Piotra.

Może być jeszcze gorzej. Rozważmy wyrażenie wykonywane w ramach procesu wypłaty:

```
(set! balance (- balance amount))
```

Jest ono wykonywane w trzech krokach: (1) pobierana jest wartość zmiennej *balance*; (2) obliczane jest nowe saldo; (3) zmiennej *balance* przypisywana

<sup>35</sup> Można tu zacytować graffiti ze ściany budynku w Cambridge: „Czas wynaleziono po to, by nie wszystko działało się jednocześnie”.



Rys. 3.29. Diagram przebiegu zdarzeń pokazujący, jak przeplot zdarzeń składających się na dwie wypłaty bankowe może prowadzić do niepoprawnego salda końcowego

jest nowa wartość. Jeśli Piotr i Paweł wykonują tę instrukcję współbieżnie, to dwie wypłaty mogą przeplatać kolejność, w jakiej pobierają i przypisują wartości zmiennej `balance`.

Diagram przebiegu zdarzeń pokazany na rys. 3.29 przedstawia kolejność zdarzeń, gdy na początku `balance` jest równe 100, Piotr wypłaca 10, Paweł wypłaca 25, a mimo to na końcu `balance` jest równe 75. Jak widać na diagramie, powodem tej anomalii jest to, że Paweł przypisuje `balance` wartość 75, zakładając, że zmniejsza wartość tej zmiennej ze 100. Jednak to założenie jest fałszywe, gdyż Piotr w tym czasie zmienił wartość `balance` na 90. Jest to katastrofalny błąd w systemie bankowym, gdyż łączna ilość pieniędzy w systemie nie jest zachowana. Łączna suma pieniędzy przed transakcjami wynosiła 100 dolarów. Po czym Piotr ma 10 dolarów, Paweł ma 25 dolarów, a w banku jest 75 dolarów<sup>36</sup>.

<sup>36</sup> W tym systemie mógłby się pojawić jeszcze poważniejszy błąd, jeśli dwie operacje próbowałyby równocześnie zmienić saldo konta, w wyniku czego faktyczne dane, które pojawią się w pamięci, mogą być dowolną kombinacją informacji zapisywanych przez obydwa procesy. W większości komputerów istnieją blokady nałożone na pierwotne operacje zapisu do pamięci, które chronią przed takim równoczesnym dostępem do pamięci. Jednakże nawet pozworne tak prosta ochrona stanowi wyzwanie implementacyjne przy konstruowaniu komputerów wieloprocesorowych, gdzie konieczny jest złożony protokół *zgodności pamięci podręcznej*, aby zapewnić, że każdy procesor będzie miał spójny obraz zawartości pamięci, bez względu na to, że dane mogą być powielane w pamięciach podręcznych wielu procesorów, umożliwiając szybszy dostęp do pamięci.

Ogólne zjawisko zilustrowane powyżej polega na tym, że wiele procesów może współdzielić zmienne stanu. Tym, co komplikuje sprawy, jest fakt, że wiele procesów może równocześnie próbować operować na współdzielonych zmiennych. W przykładzie z kontem bankowym podczas poszczególnych transakcji każdy klient powinien móc działać tak, jak gdyby nie było żadnych innych klientów. Gdy klient zmienia saldo konta w sposób, który zależy od tego salda, musi mieć możliwość założenia, że aż do momentu, w którym zmienia saldo, pozostaje ono takie, jak mu się wydaje.

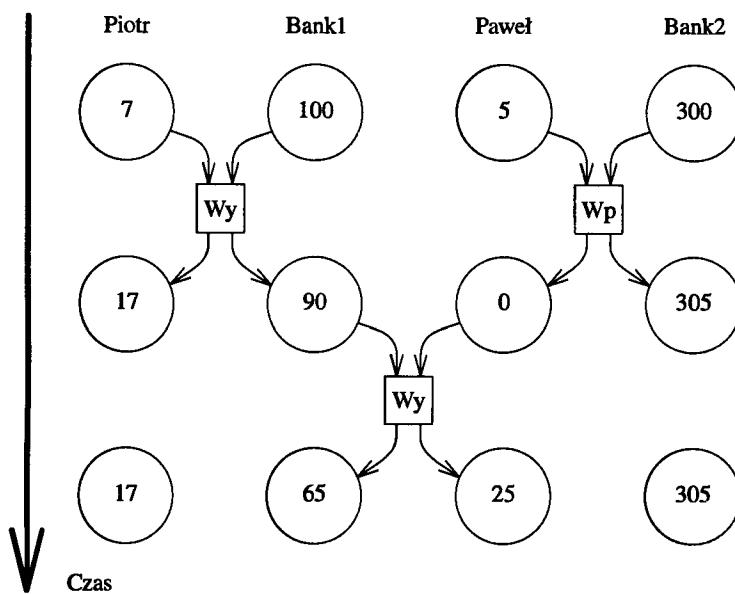
### Poprawność programów współbieżnych

Opisana powyżej sytuacja stanowi przykład subtelnych błędów, które mogą się wkrąć do programów współbieżnych. Źródło tych problemów leży w przypisaniach do zmiennych współdzielonych przez różne procesy. Wiemy już, że musimy uważać, pisząc programy korzystające z `set!`, gdyż wyniki obliczenia zależą od kolejności, w jakiej są wykonywane przypisania<sup>37</sup>. W procesach współbieżnych musimy specjalnie uważać na przypisania, ponieważ możemy nie być w stanie kontrolować kolejności, w jakiej różne procesy wykonują przypisania. Jeśli wiele takich zmian może być wykonanych współbieżnie (jak w wypadku dwóch deponentów korzystających ze wspólnego konta), to potrzebujemy jakiegoś sposobu zapewnienia, żeby nasz system zachowywał się poprawnie. Przy wypłatach ze wspólnego konta bankowego musimy na przykład zapewnić, że łączna suma pieniędzy jest zachowana. Chcąc sprawić, aby programy współbieżne działały poprawnie, możemy być zmuszeni nałożyć pewne ograniczenia na ich współbieżne wykonanie.

Jedno z możliwych ograniczeń współbieżności mogłoby narzucać, że żadne dwie operacje zmieniające współdzielone zmienne stanu nie mogą być wykonywane w tym samym czasie. Jest to niezwykle surowe wymaganie. W przypadku rozproszonych usług bankowych oznaczałoby to, że konstruktor systemu musiałby zapewnić, iż w danej chwili może być wykonywana tylko jedna transakcja. Byłoby to zarówno nieefektywne, jak i zbyt zachowawcze. Na rysunku 3.30 widać sytuację, w której Piotr i Paweł współdzielą konto bankowe, przy czym Paweł ma również swoje prywatne konto. Diagram przedstawia dwie wypłaty ze wspólnego konta (jednej dokonuje Piotr, a drugiej Paweł) i wpłatę na prywatne konto Pawła<sup>38</sup>. Obie wypłaty ze wspólnego konta nie mogą być współbieżne (gdyż obie mają dostęp i zmieniają stan tego samego konta), a także wpłata i wypłata Pawła nie mogą być współbieżne (gdyż obie mają

<sup>37</sup> Program obliczający silnię, przedstawiony w punkcie 3.1.3, ilustruje ten sam problem w przypadku jednego procesu sekwencyjnego.

<sup>38</sup> Kolumny reprezentują zawartość: portfela Piotra, wspólnego konta (Bank1), portfela Pawła oraz jego prywatnego konta (Bank2), przed i po każdej wypłacie (Wy) i wpłacie (Wp). Piotr wypłaca 10 dolarów z konta Bank1, Paweł wpłaca 5 dolarów na konto Bank2, a następnie wypłaca 25 dolarów z konta Bank1.



Rys. 3.30. Współbieżne wpłaty i wypłaty ze wspólnego konta Bank1 i prywatnego konta Bank2

dostęp i zmieniają kwotę pieniędzy znajdujących się w portfelu Pawła). Nie powinno być jednak żadnych przeciwnskazań do tego, aby wpłata Pawła na jego prywatne konto przebiegała współbieżnie z wypłatą Piotra ze wspólnego konta.

Mniej rygorystyczne ograniczenie współbieżności mogłoby gwarantować, że wyniki działania systemu współbieżnego powinny być takie same jak przy sekwencyjnym wykonaniu procesów w pewnej kolejności. Ograniczenie to ma dwie istotne cechy. Po pierwsze, nie wymaga, aby procesy faktycznie były wykonywane sekwencyjnie, a jedynie dawały te same wyniki, *jak gdyby* były wykonywane sekwencyjnie. Dla przykładu z rys. 3.30 projektant systemu kont bankowych może spokojnie pozwolić, aby wpłata Pawła i wypłata Piotra odbywały się współbieżnie, gdyż wynik końcowy jest taki sam, jak gdyby obydwie operacje były wykonane sekwencyjnie. Po drugie, może być wiele możliwych „poprawnych” wyników działania programu współbieżnego, gdyż wymagamy jedynie, aby były one takie same jak w przypadku sekwencyjnego wykonania w *pewnej* kolejności. Przypuśćmy na przykład, że na wspólnym koncie Piotra i Pawła jest na początku 100 dolarów, Piotr wpłaca 40 dolarów, a Paweł równocześnie wypłaca z konta połowę pieniędzy. W wyniku sekwencyjnego wykonania tych operacji, końcowe saldo konta może wynosić 70 lub 90 dolarów (zob. ćwiczenie 3.38)<sup>39</sup>.

<sup>39</sup> Możemy tę ideę wyrazić bardziej formalnie, mówiąc, że programy współbieżne są z natury *niedeterministyczne*. Oznacza to, że nie możemy ich opisać funkcją o jednej wartości, ale

Możliwe są jeszcze słabsze wymagania gwarantujące poprawne wykonanie programów współbieżnych. Program symulujący (powiedzmy, rozchodzenie się ciepła w ciele) może składać się z dużej liczby procesów, z których każdy reprezentuje mały obszar przestrzeni i każdy współbieżnie aktualnia swoje dane. Każdy z tych procesów wielokrotnie zmienia swoją wartość na średnią z wartości swojej i swoich sąsiadów. Taki algorytm zbiega do poprawnego wyniku niezależnie od kolejności, w jakiej są wykonywane operacje — nie ma więc powodów do żadnych ograniczeń współbieżnego wykorzystania współdzielonych wartości.

### Ćwiczenie 3.38

Przypuśćmy, że Piotr, Paweł i Maria mają wspólne konto, na którym znajduje się początkowo 100 dolarów. Piotr wpłaca 10 dolarów, Paweł wypłaca 20 dolarów, a Maria wypłaca z konta połowę pieniędzy, wykonując współbieżnie następujące polecenia:

Piotr: `(set! balance (+ balance 10))`  
 Paweł: `(set! balance (- balance 20))`  
 Maria: `(set! balance (/ balance 2)))`

- (a) Wymień wszystkie możliwe różne wartości `balance` po wykonaniu tych trzech transakcji, zakładając, że system bankowy wymusza sekwencyjne wykonanie tych operacji w pewnej kolejności.  
 (b) Podaj kilka innych wartości, jakie mogłyby powstać, gdyby system pozwalał na przeplatanie się tych poleceń. Narysuj diagramy kolejności zdarzeń, takie jak na rys. 3.29, wyjaśniające, jak te wartości mogą powstać.

#### 3.4.2. Mechanizmy sterowania współbieżnością

Jak widzieliśmy, trudności w posługiwaniu się procesami współbieżnymi wynikają z konieczności rozważania przeplotów zdarzeń zachodzących w różnych procesach. Przypuśćmy na przykład, że mamy dwa procesy; w jednym zachodzą trzy uszeregowane zdarzenia  $(a, b, c)$ , a w drugim trzy uszeregowane zdarzenia  $(x, y, z)$ . Jeśli te dwa procesy uruchomimy współbieżnie, bez żadnych ograniczeń na możliwe przeploty ich wykonania, to istnieje 20 różnych możliwych kolejności zdarzeń zgodnych z ich uszeregowaniem w procesach:

$(a, b, c, x, y, z)$   $(a, x, b, y, c, z)$   $(x, a, b, c, y, z)$   $(x, a, y, z, b, c)$   
 $(a, b, x, c, y, z)$   $(a, x, b, y, z, c)$   $(x, a, b, y, c, z)$   $(x, y, a, b, c, z)$   
 $(a, b, x, y, c, z)$   $(a, x, y, b, c, z)$   $(x, a, b, y, z, c)$   $(x, y, a, b, z, c)$   
 $(a, b, x, y, z, c)$   $(a, x, y, b, z, c)$   $(x, a, y, b, c, z)$   $(x, y, a, z, b, c)$   
 $(a, x, b, c, y, z)$   $(a, x, y, z, b, c)$   $(x, a, y, b, z, c)$   $(x, y, z, a, b, c)$

Jako programiści konstruujący ten system powinniśmy rozważyć rezultaty wszystkich 20 kolejności zdarzeń i upewnić się, że każdy z nich jest akcepto-

funkcją, której wynikiem jest zbiór możliwych wartości. W podrozdziale 4.3 zajmiemy się językiem służącym do wyrażania obliczeń niedeterministycznych.

walny. Takie podejście w miarę zwiększania liczby procesów i zdarzeń bardzo szybko staje się trudne do wykonania.

Bardziej praktyczne podejście do tworzenia systemów współbieżnych polega na opracowaniu ogólnego mechanizmu pozwalającego na ograniczanie przepłatania się procesów współbieżnych w taki sposób, że możemy być pewni, iż zachowanie programu jest poprawne. Opracowano wiele takich mechanizmów. W niniejszym punkcie opiszemy jeden z nich — *sekcje krytyczne*<sup>\*</sup>.

### Szeregowanie dostępu do współdzielonych zmiennych stanu

Sekcje krytyczne stanowią implementację następującego pomysłu: Procesy są wykonywane współbieżnie, istnieją jednak zestawy procedur, które nie mogą być wykonywane współbieżnie. Dokładniej, sekcje krytyczne to wyróżnione zbiory takich procedur, że tylko jedna procedura z każdego takiego zbioru może być wykonywana w danej chwili. Jeśli jedna procedura ze zbioru jest właśnie wykonywana, a jakiś proces próbuje wykonać dowolną procedurę z tego samego zbioru, to musi on zaczekać, aż pierwsza procedura zostanie wykonana.

Możemy użyć sekcji krytycznych do sterowania dostępem do zmiennych współdzielonych. Jeśli na przykład chcemy zmodyfikować wartość zmiennej współdzielonej w sposób, który zależy od poprzedniej wartości tej zmiennej, to umieszczamy pobranie poprzedniej wartości zmiennej i przypisanie nowej wartości do zmiennej w jednej procedurze. Następnie zapewniamy, że żadna inna procedura przypisująca wartość tej zmiennej nie może być wykonywana równocześnie z naszą procedurą, umieszczając wszystkie takie procedury w jednej sekcji krytycznej. Gwarantuje to, że wartość zmiennej, między jej pobraniem a przypisaniem, nie może ulec zmianie.

### Sekcje krytyczne w języku Scheme

Aby ukonkretnić powyższy mechanizm, przyjmijmy, że rozszerzamy język Scheme o procedurę o nazwie `parallel-execute`:

`(parallel-execute <p1> <p2> ... <pk>)`

Każde `<p>` musi być procedurą bezargumentową. `Parallel-execute` tworzy dla każdego `<p>` osobny proces, który wywołuje `<p>` (bez argumentów). Wszystkie te procesy działają współbieżnie<sup>40</sup>.

\* Autorzy używają terminu *serializer*, który nie ma dobrego odpowiednika w terminologii polskiej. Używamy tu terminu „sekcja krytyczna”, gdyż najlepiej odpowiada znaczeniu opisanego mechanizmu (przyp. tłum.).

<sup>40</sup> Procedura `parallel-execute` nie jest częścią standardu języka Scheme, ale można ją zaimplementować w MIT Scheme. W naszej implementacji nowe procesy współbieżne działają równolegle z oryginalnym procesem Scheme'a. Ponadto w naszej implementacji wynikiem `parallel-execute` jest specjalny obiekt sterujący, którego można użyć do zatrzymania nowo utworzonych procesów.

Rozważmy następujący przykład użycia tej procedury:

```
(define x 10)

(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (+ x 1))))
```

Tworzone są dwa procesy współbieżne —  $P_1$ , który przypisuje zmiennej  $x$  wartość  $x$  razy  $x$ , oraz  $P_2$ , który zwiększa  $x$  o 1. Po ich wykonaniu  $x$  może mieć pięć różnych wartości, zależnie od przeplotu zdarzeń procesów  $P_1$  i  $P_2$ :

- 101:  $P_1$  przypisuje  $x$  wartość 100, a następnie  $P_2$  zwiększa  $x$  do 101.
- 121:  $P_2$  zwiększa  $x$  do 11, a następnie  $P_1$  podnosi  $x$  do kwadratu.
- 110:  $P_2$  zwiększa  $x$  z 10 do 11 pomiędzy dwoma pobraniemiami wartości  $x$  przez  $P_1$  w trakcie obliczania  $(* x x)$ .
- 11:  $P_2$  pobiera wartość  $x$ , następnie  $P_1$  przypisuje  $x$  wartość 100, po czym  $P_2$  przypisuje wartość  $x$ .
- 100:  $P_1$  (dwukrotnie) pobiera wartość  $x$ , następnie  $P_2$  przypisuje  $x$  wartość 11, po czym  $P_1$  przypisuje wartość  $x$ .

Możemy ograniczyć współbieżność, używając sekcji krytycznych. Sekcje krytyczne są tworzone przez procedurę `make-serializer`, której implementację podajemy dalej. Argumentem sekcji krytycznej jest procedura, a wynikiem jest tak samo działająca procedura, ale należąca do tej sekcji krytycznej. Wynikami wszystkich wywołań danej sekcji krytycznej są procedury należące do tej samej sekcji krytycznej.

Tak więc, w odróżnieniu od powyższego przykładu, wykonanie

```
(define x 10)

(define s (make-serializer))

(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (+ x 1)))))
```

może dać w wyniku tylko dwie wartości  $x$ : 101 lub 121. Inne możliwości zostały wyeliminowane, gdyż wykonania procesów  $P_1$  i  $P_2$  nie mogą się przeplatać.

Oto wersja procedury `make-account` z punktu 3.1.1, w której wpłaty i wyплатy zostały umieszczone w jednej sekcji krytycznej:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie"))
```

```
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(let ((protected (make-serializer)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) (protected withdraw))
          ((eq? m 'deposit) (protected deposit))
          ((eq? m 'balance) balance)
          (else (error "Nieznana operacja -- MAKE-ACCOUNT"
                       m))))
  dispatch))
```

W tej implementacji dwa procesy nie mogą równocześnie wpłacać lub wypłacać pieniędzy z tego samego konta. Usuwa to przyczynę błędu pokazanego na rys. 3.29, gdzie Piotr zmienia saldo konta między tym, jak Paweł pobiera saldo konta, żeby obliczyć jego nową wartość, i tym, jak przypisuje saldu tę nową wartość. Z drugiej strony, każde konto ma swoją sekcję krytyczną, a więc wpłaty i wypłaty dotyczące różnych kont mogą odbywać się równocześnie.

### Ćwiczenie 3.39

Które z pięciu pokazanych wcześniej możliwości równoległego wykonania mogą zajść, jeżeli sekcji krytycznej użyjemy w następujący sposób:

```
(define x 10)

(define s (make-serializer))

(parallel-execute (lambda () (set! x ((s (lambda () (* x x)))))))
                  (s (lambda () (set! x (+ x 1)))))
```

### Ćwiczenie 3.40

Podaj wszystkie możliwe wartości x, które mogą być wynikami wykonania

```
(define x 10)

(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (* x x x))))
```

Które z tych możliwości mogą zajść, jeżeli w powyższych procedurach użyjemy sekcji krytycznej:

```
(define x 10)

(define s (make-serializer))

(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (* x x x)))))
```

### Ćwiczenie 3.41

Ben Bajerbit martwi się, że może lepiej byłoby zaimplementować konto bankowe w następujący sposób (komentarzem zaznaczono wiersz, który został zmieniony):

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance)
             ((protected (lambda () balance)))) ; dodana sekcja krytyczna
            (else (error "Nieznana operacja -- MAKE-ACCOUNT"
                         m))))
    dispatch)))

```

gdź umozliwienie pobierania salda konta w dowolnym momencie może powodować nieprawidłowe działanie. Czy zgadzasz się z tym? Czy istnieje jakikolwiek scenariusz potwierdzający obawy Bena?

### Ćwiczenie 3.42

Ben Bajerbit zauważył, że stratą czasu jest umieszczanie nowej procedury w sekcji krytycznej w odpowiedzi na każdy komunikat `withdraw` i `deposit`. Mówi, że można by tak zmienić `make-account`, aby wywołania procedury `protected` odbywały się na zewnątrz procedury `dispatch`. Oznacza to, że konto przekazywałoby nam tę samą procedurę należącą do sekcji krytycznej (utworzoną w tym samym momencie co konto), za każdym razem gdy poprosimy o procedurę wypłaty.

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (let ((protected-withdraw (protected withdraw))
          (protected-deposit (protected deposit)))
      (define (dispatch m)
        (cond ((eq? m 'withdraw) protected-withdraw)
              ((eq? m 'deposit) protected-deposit)
              (else (error "Nieznana operacja -- MAKE-ACCOUNT"
                           m)))))))

```

```
((eq? m 'balance) balance)
(else (error "Nieznana operacja -- MAKE-ACCOUNT"
m)))
dispatch)))
```

Czy można bezpiecznie dokonać takiej zmiany? W szczególności, czy ma to jakikolwiek wpływ na możliwe współbieżne wykonania tych dwóch wersji `make-account`?

### Złożoność korzystania z wielu zasobów współdzielonych

Sekcje krytyczne stanowią silną abstrakcję, która pomaga rozdzielić złożoność programów współbieżnych tak, aby każdy z nich był traktowany z należytą uwagą i (miejmy nadzieję) był poprawny. Chociaż używanie sekcji krytycznych jest stosunkowo proste, gdy jest tylko jeden zasób współdzielony (taki jak jedno konto bankowe), programowanie współbieżne może być jednak zdradliwe i skomplikowane, gdy mamy do czynienia z wieloma zasobami współdzielonymi.

Aby zilustrować jeden z rodzajów trudności, jakie mogą powstać, przyjmijmy, że chcemy zamienić miejscami salda dwóch kont bankowych. Pobieramy saldo każdego konta, obliczamy ich różnicę, wypłacamy różnicę z jednego konta i wpłacamy ją na drugie konto. Moglibyśmy zaimplementować to w następujący sposób<sup>41</sup>:

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                        (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

Procedura ta działa dobrze wtedy, gdy tylko jeden proces próbuje dokonać zamiany. Przypuśćmy jednak, że zarówno Piotr, jak i Paweł mają dostęp do kont *a1*, *a2* i *a3* oraz że Piotr zamienia *a1* z *a2*, podczas gdy Paweł równocześnie zamienia *a1* z *a3*. Nawet jeśli umieścimy operacje wpłaty i wypłaty w sekcjach krytycznych związanych z poszczególnymi kontami (tak jak w procedurze `make-account` pokazanej wcześniej w niniejszym podpunkcie), to procedura `exchange` może nadal dawać złe wyniki. Piotr na przykład może obliczyć różnicę między saldami *a1* i *a2*, ale Paweł może zmienić saldo *a1*, zanim Piotr zdąży dokończyć zamianę<sup>42</sup>. Żeby procedura `exchange` działa-

<sup>41</sup> Uprościliśmy procedurę `exchange`, wykorzystując fakt, że komunikatowi `deposit` może towarzyszyć ujemna kwota. (Jest to poważny błąd w naszym systemie bankowym!)

<sup>42</sup> Jeśli na trzech kontach znajduje się początkowo, odpowiednio, 10, 20 i 30 dolarów, to po dowolnej liczbie współbieżnych zamian salda kont w pewnej kolejności powinny nadal wynosić 10, 20 i 30 dolarów. Umieszczenie operacji wpłaty w sekcji krytycznej nie gwarantuje nam tego. Zobacz ćwiczenie 3.43.

ła poprawnie, musimy zapewnić, że jej wykonanie przez cały czas trwania zamiany wyklucza jakikolwiek inny współbieżny dostęp do obu kont.

Można to zrobić, umieszczaając całą procedurę `exchange` w dwóch sekcjach krytycznych związkanych z obydwoma kontami. Aby tego dokonać, umożliwimy dostęp do sekcji krytycznej konta. Zauważmy, że tym samym celowo łamimy modularność obiektu konta bankowego, eksponując jego sekcję krytyczną. Następująca wersja `make-account` jest identyczna z pierwotną wersją z punktu 3.1.1 z wyjątkiem tego, że utworzono sekcję krytyczną do ochrony salda i udostępniono ją poprzez mechanizm przekazywania komunikatów:

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Brak środków na koncie"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Nieznaną operację -- MAKE-ACCOUNT"
                         m)))))

  dispatch))
```

Możemy jej użyć do szeregowania wpłat i wypłat. W odróżnieniu od poprzedniej wersji konta z szeregowaniem operacji, tutaj do użytkownika należy jawne szeregowanie operacji, na przykład w następujący sposób<sup>43</sup>:

```
(define (deposit account amount)
  (let ((s (account 'serializer)))
    (d (account 'deposit)))
  ((s d) amount)))
```

Eksportując na zewnątrz sekcję krytyczną, dysponujemy wystarczającymi środkami do zaimplementowania programu zamiany sald z szeregowaniem. Umieszczały po prostu procedurę `exchange` w sekcjach krytycznych związanych z obydwoma kontami:

---

<sup>43</sup> W ćwiczeniu 3.45 zajmujemy się tym, dlaczego wpłaty i wypłaty nie są już automatycznie szeregowane przez konto.

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))
```

### Ćwiczenie 3.43

Przypuśćmy, że mamy trzy konta, których salda wynoszą 10, 20 i 30 dolarów, i że wiele współbieżnie działających procesów zamienia miejscami salda kont. Uzasadnij, że gdyby procesy działały sekwencyjnie, to po dowolnej liczbie równoczesnych zamian salda kont w pewnej kolejności powinny nadal wynosić 10, 20 i 30 dolarów. Narysuj diagram kolejności zdarzeń, taki jak ten na rys. 3.29, pokazujący, że ten warunek nie musi być spełniony, jeżeli procesy będą używać procedury zamiany sald przedstawionej jako pierwsza w niniejszym punkcie. Uzasadnij, że z drugiej strony nawet ta wersja programu exchange zachowa niezmienioną sumę sald wszystkich kont. Narysuj diagram kolejności zdarzeń pokazujący, że nawet ten warunek mógłby nie być spełniony, gdyby operacje na jednym koncie nie były szeregowane.

### Ćwiczenie 3.44

Rozważmy problem przelewania pieniędzy z jednego konta na drugie. Ben Bajerbit twierdzi, że można zrealizować przelew za pomocą poniższej procedury, nawet jeśli wiele osób będzie równocześnie przelewać pieniądze między wieloma kontami, używając dowolnego mechanizmu szeregowania wpłat i wypłat, na przykład przedstawionej powyżej wersji `make-account`.

```
(define (transfer from-account to-account amount)
  ((from-account 'withdraw) amount)
  ((to-account 'deposit) amount))
```

Ludwik Myślicielak twierdzi, że istnieje tu pewien problem i że powinno się użyć bardziej wyszukanej metody, takiej jak ta zastosowana w przypadku zamiany sald. Czy Ludwik ma rację? Jeśli nie, to jaka jest zasadnicza różnica między przelewem a zamianą sald? (Powinieneś założyć, że saldo konta `from-account` jest nie mniejsze niż `amount`).

### Ćwiczenie 3.45

Ludwik Myślicielak uważa, że z chwilą gdy operacje wpłat i wypłat przestaną być automatycznie szeregowane, system kont bankowych stał się niepotrzebnie skomplikowany i podatny na błędy. Sugeruje on, że procedura `make-account-and-serializer` powinna eksportować sekcję krytyczną (na potrzeby takich operacji jak `serialized-exchange`) oprócz (a nie zamiast) użycia jej do szeregowania wpłat i wypłat, tak jak to ma miejsce w procedurze `make-account`. Proponuje on, aby przedefiniować konta w następujący sposób:

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
```

```

(if (>= balance amount)
  (begin (set! balance (- balance amount))
         balance)
  "Brak środków na koncie"))
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(let ((balance-serializer (make-serializer)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) (balance-serializer withdraw))
          ((eq? m 'deposit) (balance-serializer deposit))
          ((eq? m 'balance) balance)
          ((eq? m 'serializer) balance-serializer)
          (else (error "Nieznana operacja -- MAKE-ACCOUNT"
                       m))))
  dispatch))

```

Wówczas wpłaty są wykonywane tak, jak w przypadku oryginalnej procedury `make-account`:

```

(define (deposit account amount)
  ((account 'deposit) amount))

```

Wyjaśnij, gdzie w rozumowaniu Ludwika tkwi błąd. W szczególności rozważ, co się stanie, gdy zostanie wywołana procedura `serialized-exchange`.

### Implementowanie sekcji krytycznej

Sekcje krytyczne implementujemy za pomocą bardziej elementarnego mechanizmu synchronizacji nazywanego *semaphorem*. Semafor to obiekt, na którym można wykonywać dwie operacje — semafor można *opuszczać* (ang. *acquire*) i *podnosić* (ang. *release*). Gdy opuścimy semafor, żadna inna operacja opuszczania semafora nie zostanie wykonana, dopóki go nie podniesiemy<sup>44</sup>. W naszej implementacji z każdą sekcją krytyczną jest związany semafor. Dla danej procedury p wynikiem umieszczenia jej w sekcji krytycznej jest procedura, która opuszcza semafor, wykonuje p, a następnie podnosi semafor. Zapewnia to, że tylko jedna procedura należąca do danej sekcji krytycznej

<sup>44</sup> Angielska nazwa semafora „*mutex*” jest skrótem od *mutual exclusion* (wzajemnego wykluczania). Terminem tym określa się ogólny problem konstruowania mechanizmu, który pozwala procesom współbieżnym bezpiecznie współdzielić zasoby [tzn. w taki sposób, że w danej chwili do każdego zasobu ma dostęp co najwyżej jeden proces; przyp. tłum.]. Nasze semafory są uproszczoną wersją *semafora ogólnego*, nazywanego semaforem binarnym (zob. ćwiczenie 3.47), wprowadzonego w systemie operacyjnym „THE”, opracowanym w Technische Hogeschool w Eindhoven, w Holandii, i nazwanym inicjalami uczelni [18]. Oryginalnie operacje opuszczenia i podniesienia semafora nazywały się P i V, od holenderskich słów *passeren* (przechodzić) i *vrijgeven* (zwalniać) — w nawiązaniu do semaforów kolejowych. [Według innych źródeł nazwy tych operacji pochodzą od holenderskich słów *proberen* (testować) i *verhogen* (zwiększać); przyp. tłum.]. Klasyczne już wystąpienie Dijkstry [19] było pierwszym, w którym przejrzyście przedstawiono problemy sterowania współbieżnością i pokazano, jak można rozwiązać różne problemy współbieżne za pomocą semaforów.

może być wykonywana w danej chwili, co jest dokładnie tym, co chcieliśmy zagwarantować.

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
      serialized-p)))
```

Semafor jest obiektem modyfikowalnym (będziemy tu używać jednoelementowych list, które będziemy nazywać *komórkami* (ang. *cells*)) mogącym przechowywać dwie wartości: prawdę lub fałsz. Gdy wartość jest fałszem, semafor jest dostępny (podniesiony) i można go opuścić. Gdy wartość jest prawdą, semafor jest niedostępny (opuszczony) i każdy proces, który próbuje go opuścić, musi czekać.

Konstruktor semaforów `make-mutex` inicjuje komórkę, zapamiętując w niej wartość fałsz. Chcąc opuścić semafor, sprawdzamy zawartość komórki. Jeśli semafor jest podniesiony, zapisujemy w komórce wartość prawda i kontynuujemy. W przeciwnym razie czekamy w pętli, próbując raz po razie opuścić semafor, aż nam się to uda<sup>45,\*</sup>. Chcąc podnieść semafor, zapisujemy w komórce wartość fałsz.

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
              (if (test-and-set! cell)
                  (the-mutex 'acquire))) ; spróbuj ponownie
            ((eq? m 'release) (clear! cell)))
            the-mutex))

  (define (clear! cell)
    (set-car! cell false)))
```

<sup>45</sup> W większości systemów operacyjnych z podziałem czasu procesy zablokowane przez semafor nie marnują czasu, jak wyżej, na *aktywne oczekiwanie* (ang. *busy waiting*). Zamiast tego, podczas gdy dany proces czeka, system uruchamia inny proces, a kiedy semafor zostaje podniesiony, system „budzi” czekający proces.

\* Taka implementacja operacji na semaforze nie zapewnia jednej cechy, jaką powinien on mieć — uczciwości. Jeśli semafor jest podnoszony dostatecznie wiele razy, to każdy proces czekający na jego podniesienie powinien się tego doczekać. Tymczasem możemy sobie wyobrazić sytuację, w której kilka procesów na przemian opuszcza i podnosi semafor. Część procesów oczekujących na podniesienie semafora, jeżeli będzie miała „nieskończonego pecha”, nigdy się tego nie doczeka, gdyż zawsze inny czekający proces wykorzysta podniesienie semafora. Sytuację taką nazywamy *zagłodzeniem* (ang. *starvation*) (przyp. tłum.).

Operacja sprawdź-i-ustaw (`test-and-set!`) sprawdza zawartość komórki i przekazuje ją jako wynik. Dodatkowo, jeśli komórka zawiera wartość fałsz, operacja ta zapisuje w niej wartość prawda, po czym przekazuje wartość fałsz jako wynik. Jej działanie możemy wyrazić w postaci następującej procedury:

```
(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

Jednakże taka implementacja `test-and-set!` jest niewystarczająca. Kryje się tu subtelna, acz istotna kwestia, zasadnicza dla sterowania współbieżnością w systemie — operacja sprawdź-i-ustaw musi być operacją *atomową* (niepodzielną). Oznacza to, iż musimy zagwarantować, że jeżeli proces zbada wartość komórki i będzie ona fałszem, to w komórce zostanie faktycznie zapisana prawda, zanim jakkolwiek inny proces zbada zawartość komórki. Jeżeli tego nie zagwarantujemy, to semafor może zawieść; podobnie jak konto bankowe zawiodło w sposób przedstawiony na rys. 3.29. (Zobacz ćwiczenie 3.46).

Faktyczna implementacja `test-and-set!` zależy od szczegółów tego, jak system wykonuje procesy współbieżne. Mogą one na przykład być wykonywane przez procesor sekwencyjnie przy użyciu mechanizmu kwantowania czasu, który polega na przydzielaniu procesora procesowi na krótki okres czasu, po czym działanie procesu jest przerywane i procesor jest przydzielany kolejnemu procesowi. W takim przypadku `test-and-set!` może wstrzymać mechanizm kwantowania czasu na czas sprawdzenia i ustawienia komórki<sup>46</sup>. Alternatywnie, w komputerach wieloprocesorowych są dostępne instrukcje (rozkazy) wspierające tworzenie operacji atomowych, zaimplementowane bezpośrednio sprzętowo<sup>47</sup>.

<sup>46</sup> W implementacji MIT Scheme dla maszyny jednoprocesorowej, która stosuje kwantowanie czasu, operację `test-and-set!` można zaimplementować tak:

```
(define (test-and-set! cell)
  (without-interrupts
  (lambda ()
    (if (car cell)
        true
        (begin (set-car! cell true)
               false)))))
```

Operacja `without-interrupts` wyłącza mechanizm kwantowania czasu na czas wykonania jej argumentów.

<sup>47</sup> Istnieje wiele wariantów takich instrukcji — wliczając w to instrukcję sprawdź-i-ustaw (ang. *test-and-set*), instrukcję sprawdź-i-wyzeruj (ang. *test-and-clear*), instrukcję zamiany (ang. *swap*), instrukcję porównaj-i-zamień (ang. *compare-and-exchange*), instrukcję ładuj-i-rezerwuj (ang. *load-and-reserve*) oraz instrukcję warunkowo-zapisz (ang. *store-conditional*) — których

### Ćwiczenie 3.46

Przypuśćmy, że zaimplementujemy operację **test-and-set!** za pomocą zwykłych procedur, nie starając się uczynić z niej operacji atomowej. Narysuj diagram kolejności zdarzeń, taki jak na rys. 3.29, pokazujący jak zawodzi implementacja semafora, pozwalając dwóm procesom w tym samym czasie opuścić ten sam semafor.

### Ćwiczenie 3.47

Semafor (rozmiaru  $n$ ) stanowi uogólnienie semafora binarnego. Tak jak semafor binarny udostępnia on operacje podnoszenia i opuszczania, ale jest o tyle bardziej ogólny, że może być podniesiony na wysokość od 0 do  $n$ , czyli może być opuszczony równocześnie przez maksymalnie  $n$  procesów. Dodatkowe procesy próbujące opuścić semafor muszą czekać na podniesienie go. Podaj implementację semaforów rozmiaru  $n$ :

- (a) za pomocą semaforów binarnych,
- (b) za pomocą operacji atomowych **test-and-set!**.

### Zakleszczenie

Teraz, gdy już zobaczyliśmy, jak można zaimplementować sekcję krytyczną, widzimy, że nadal jest problem z zamianą sald kont, nawet jeśli użyjemy przedstawionej wcześniej procedury **serialized-exchange**. Wyobraźmy sobie, że Piotr stara się zamienić  $a1$  z  $a2$ , podczas gdy Paweł równocześnie stara się zmienić  $a2$  z  $a1$ . Przypuśćmy, że proces Piotra dochodzi do miejsca, w którym znajduje się już w sekcji krytycznej chroniącej  $a1$ , a zaraz po tym proces Pawła wchodzi do sekcji krytycznej chroniącej  $a2$ . Wówczas proces Piotra nie może kontynuować działania (wejść do sekcji krytycznej chroniącej  $a2$ ), dopóki proces Pawła nie wyjdzie z sekcji krytycznej chroniącej  $a2$ . Podobnie proces Pawła nie może kontynuować działania, dopóki proces Piotra nie wyjdzie z sekcji krytycznej chroniącej  $a1$ . Każdy z procesów jest zablokowany, czekając w nieskończoność na drugi proces. Sytuację taką nazywamy **zakleszczeniem** (ang. *deadlock*). Zakleszczenie jest niebezpieczeństwem, które zawsze występuje w systemach udostępniających wspólnie wiele zasobów dzielonych.

Jeden ze sposobów uniknięcia zakleszczenia w takiej sytuacji polega na przydzieleniu każdemu kontu niepowtarzalnego numeru identyfikacyjnego i takim przepisaniu procedury **serialized-exchange**, aby procesy starały się

---

realizacja musi być starannie dopasowana do interfejsu procesor-pamięć. Jedna z kwestii polega na ustaleniu, co się dzieje, jeśli dwa procesy za pomocą takiej instrukcji próbują dokładnie w tym samym momencie zająć ten sam zasób. Potrzebny jest tu jakiś mechanizm rozstrzygający, który proces zdobywa zasób. Mechanizm taki jest nazywany **arbitem**. Arbiter zwykle sprowadza się do pewnego rodzaju urządzenia sprzętowego. Niestety, można pokazać, że skonstruowanie uczciwego arbitra działającego w 100% przypadków jest fizycznie niemożliwe, chyba że dopuścimy, aby arbiter mógł dowolnie długo się namyślać. Podstawowe zjawisko kryjące się tutaj zostało pierwotnie zaobserwowane przez XIV-wiecznego francuskiego filozofa Jeanan Buridana w jego komentarzu do *De caelo* („O niebie”) Arystotelesa. Buridan argumentuje, że idealnie racjonalny pies znajdujący się między dwoma tak samo atrakcyjnymi źródłami pożywienia zdechnie z głodu, gdyż nie będzie mógł się zdecydować, do którego z nich podejść najpierw.

zawsze wejść najpierw do sekcji krytycznej konta o najmniejszym numerze. Chociaż metoda ta rozwiązuje problem zamiany, istnieją sytuacje wymagające bardziej wyrafinowanych technik unikania zakleszczenia lub takie, w których uniknięcie zakleszczenia w ogóle nie jest możliwe. (Zobacz ćwiczenia 3.48 i 3.49)<sup>48</sup>.

### Ćwiczenie 3.48

Wyjaśnij szczegółowo, dlaczego opisana powyżej metoda unikania zakleszczenia (polegająca na tym, że konta są ponumerowane i każdy proces najpierw opuszcza semafor konta o najmniejszym numerze) pozwala uniknąć zakleszczenia przy zamianie sald. Zdefiniuj ponownie `serialized-exchange`, stosując tę metodę. (Będziesz również musiał zmodyfikować procedurę `make-account` tak, aby każde konto miało nadawany numer, który można odczytać, wysyłając odpowiedni komunikat).

### Ćwiczenie 3.49

Podaj scenariusz, w którym powyższy mechanizm unikania zakleszczenia nie zadziała. (Wskazówka: W opisany wcześniej problemie zamiany każdy proces wie z góry, do jakich kont będzie musiał uzyskać dostęp. Rozważ sytuację, w której proces musi uzyskać dostęp do pewnego zasobu dzielonego, zanim będzie wiedział, jakie jeszcze zasoby dzielone będą mu potrzebne).

## Współbieżność, czas i komunikacja

Przekonaliśmy się, że programowanie systemów współbieżnych wymaga kontroli kolejności zdarzeń, gdy różne procesy mają dostęp do wspólnych zmiennych, oraz widzieliśmy, jak taką kontrolę można zrealizować, rozważając korzystając z sekcji krytycznych. Jednak problemy współbieżności mają głębsze korzenie, gdyż z zasadniczego punktu widzenia nie zawsze jest jasne, co rozumiemy przez „współdzielenie zmiennych stanu”.

Mechanizmy takie jak `test-and-set!` wymagają od procesów sprawdzania globalnych współdzielonych znaczników w przypadkowych momentach. Jest to problematyczne i nieefektywne w implementacji dla nowoczesnych, szybkich procesorów, gdzie ze względu na takie techniki optymalizacji, jak przetwarzanie potokowe i pamięć podręczna, zawartość pamięci nie zawsze musi być spójna. Dlatego też we współczesnych systemach wieloprocesorowych paradygmat sekcji krytycznych jest wypierany przez nowe podejścia do sterowania współbieżnością<sup>49</sup>.

<sup>48</sup> Ogólna technika unikania zakleszczenia przez numerowanie zasobów dzielonych i zajmowanie ich w rosnącej kolejności numeracji pochodzi od Havendera [45]. Sytuacje, w których nie można uniknąć zakleszczenia, wymagają użycia metod *likwidowania zakleszczenia*, które oznaczają „wycofywanie” procesów ze stanu zakleszczenia i ponowną próbę ich wykonania. Mechanizmy likwidowania zakleszczenia są szeroko stosowane w systemach zarządzania bazami danych — szczegółowe omówienie tej tematyki można znaleźć w pracy Graya i Reutera [35].

<sup>49</sup> Jedna z takich alternatyw sekcji krytycznych jest nazywana *barierą synchronizacyjną* (ang. *synchronization barrier*). Programista pozwala procesom współbieżnym wykonywać się jak

Problematyczne aspekty współdzielenia zmiennych stanu widać również w dużych systemach rozproszonych. Wyobraźmy sobie na przykład rozproszony system bankowy, w którym poszczególne oddziały banku pamiętają lokalnie wartości opisujące salda bankowe i okresowo porównują je z wartościami pamiętanymi w innych oddziałach. W takim systemie wartość „salda konta” byłaby nieokreślona z wyjątkiem momentu tuż po synchronizacji. Jeśli Piotr wpłaci pieniądze na wspólne konto z Pawłem, to w którym momencie powinniśmy powiedzieć, że saldo konta uległo zmianie — gdy zmienia się saldo w lokalnym oddziale, czy też dopiero po synchronizacji? A jeśli Paweł korzysta z konta poprzez inny oddział, to jakie rozsądne ograniczenia należy nałożyć na system bankowy, aby jego działanie było „poprawne”? Jedyne, co może mieć wpływ na poprawność, to zachowanie obserwowane indywidualnie przez Piotra i Pawła oraz „stan” konta bezpośrednio po synchronizacji. Pytania dotyczące „rzeczywistego” salda konta lub kolejności zdarzeń między poszczególnymi momentami synchronizacji mogą być nieistotne lub pozbawione sensu<sup>50</sup>.

Podstawowe zjawisko polega na tym, że synchronizacja różnych procesów, utworzenie współdzielonych zmiennych stanu lub narzucenie kolejności zdarzeń wymaga komunikacji między procesami. W gruncie rzeczy każde pojęcie czasu w sterowaniu współbieżnością musi być nierozerwalnie związane z komunikacją<sup>51</sup>. Intrygujący jest fakt, że podobny związek z komunikacją występuje w teorii względności, gdzie prędkość światła (najszybszego sygnału, jaki może być użyty do synchronizacji zdarzeń) jest podstawową stałą wiążącą czas i przestrzeń. Problemy, jakie napotykamy, borykając się z czasem i stanami w naszych modelach obliczeniowych, mogą w rzeczywistości odzwierciedlać podstawowe problemy materialnego wszechświata.

### 3.5. Strumienie

Rozumiemy już dobrze rolę przypisania jako narzędzia modelowania, a także zdajemy sobie sprawę ze złożoności problemów, jakie przypisanie stwarza. Czas zadać sobie pytanie, czy nie można by postępować inaczej, tak aby

---

chć, ale wyznacza pewne punkty synchronizacyjne („bariery”), przez które żaden proces nie może przejść, dopóki wszystkie procesy nie osiągną bariery. Nowoczesne procesory udostępniają instrukcje maszynowe, umożliwiające programistom zakładać punkty synchronizacyjne w miejscach, w których wymagana jest spójność. Na przykład PowerPC™ udostępnia w tym celu instrukcje o nazwach SYNC i EIEIO (wymuszone uporządkowane wykonanie wejścia/wyjścia; ang. *Enforced In-order Execution of Input/Output*).

<sup>50</sup> Taki punkt widzenia może się wydawać dziwny, ale istnieją systemy działające w ten sposób. Na przykład międzynarodowe opłaty wykonywane za pomocą kart kredytowych są zwykle rozliczane w poszczególnych krajach, a rozliczenia dotyczące różnych krajów są okresowo uzgadniane. Tak więc saldo konta może być różne w różnych krajach.

<sup>51</sup> Takie spojrzenie na systemy rozproszone zostało zrealizowane przez Lamporta [64], który pokazał, jak można zastosować komunikację do wyznaczenia „globalnych zegarów”, których można użyć do ustalenia kolejności zdarzeń w systemie rozproszonym.

uniknąć części z tych problemów. W niniejszym podrozdziale zbadamy inne podejście do modelowania stanów, oparte na strukturach danych nazywanych *strumieniami* (ang. *streams*). Jak zobaczymy, strumienie mogą łagodzić niektóre z problemów związanych z modelowaniem stanów.

Cofnijmy się, żeby uzmysłowić sobie, z czego wynika złożoność tych problemów. Starając się modelować zjawiska ze świata rzeczywistego, podjęliśmy kilka najwyraźniej rozsądnych decyzji. Modelowaliśmy obiekty ze świata rzeczywistego ze stanami lokalnymi za pomocą obiektów obliczeniowych ze zmiennymi lokalnymi. Utożsamiliśmy bieg czasu w świecie rzeczywistym z biegiem czasu w komputerze. Zaimplementowaliśmy w komputerze zachodzące w czasie zmiany stanów modelowanych obiektów za pomocą przypisów do zmiennych lokalnych obiektów modelujących.

Czy jest możliwe inne podejście? Czy możemy uniknąć utożsamienia czasu w komputerze z czasem w modelowanym świecie? Czy nasz model musi się zmieniać w czasie, aby mógł modelować zjawiska zachodzące w zmieniającym się świecie? Pomyślmy o tej kwestii w kategoriach funkcji matematycznych. Możemy opisać zmieniające się w czasie zachowanie wielkości  $x$  jako funkcję czasu  $x(t)$ . Jeśli skoncentrujemy się na wielkości  $x$  w kolejnych momentach, to będziemy myśleć o niej jak o wielkości zmiennej. Jeśli jednak skupimy się na całej historii tej wielkości w czasie, nie będziemy uwypuklać zmiany — funkcja jako taka nie zmienia się<sup>52</sup>.

Jeśli czas jest mierzony dyskretnie, to możemy modelować wielkości będące funkcjami czasu jako (potencjalnie nieskończone) ciągi. W niniejszym podrozdziale zobaczymy, jak można modelować zmiany za pomocą ciągów reprezentujących historie modelowanych systemów. Aby tego dokonać, wprowadzimy nowe struktury danych nazywane *strumieniami*. Z abstrakcyjnego punktu widzenia strumień jest po prostu ciągiem. Przekonamy się jednak, że prosta implementacja strumieni za pomocą list (jak w punkcie 2.2.1) nie oddaje w pełni siły przetwarzania strumieni. Wprowadzimy jako alternatywę technikę *odraczania obliczeń*, która pozwala reprezentować w postaci strumieni bardzo długie (a nawet nieskończone) ciągi.

Przetwarzanie strumieni umożliwia modelowanie systemów ze stanami bez używania przypisania czy danych modyfikowalnych. Ma to ważne konsekwencje zarówno teoretyczne, jak i praktyczne, gdyż daje możliwość budowania modeli pozbawionych wad właściwych dla przypisania. Jednakże zastosowanie strumieni wiąże się z innymi problemami i kwestią wyboru techniki modelowa-

---

<sup>52</sup> Fizycy czasami przyjmują takie podejście, wprowadzając pojęcie „linii świata” cząstki jako narzędzia do wnioskowania o ruchu. Wspomnialiśmy już również (w punkcie 2.2.3), że jest to naturalny sposób myślenia o systemach przetwarzania sygnałów. Zastosowanie strumieni do przetwarzania sygnałów zbadamy w punkcie 3.5.3.

nia, która prowadzi do tworzenia systemów bardziej modularnych i łatwiejszych w utrzymaniu, pozostaje otwarta.

### 3.5.1. Strumienie jako listy odroczone

Jak widzieliśmy w punkcie 2.2.3, ciągi mogą służyć jako standardowe interfejsy łączące moduły programu. Sformułowaliśmy też potężne abstrakcje przeznaczone do operowania na ciągach, takie jak `map`, `filter` i `accumulate`, ujmujące w zwięzły i elegancki sposób najrozmaitsze operacje.

Niestety, jeśli reprezentujemy ciągi w postaci list, ta elegancja jest okupiona poważną nieefektywnością obliczeń zarówno pod względem kosztu czasowego, jak i pamięciowego. Gdy reprezentujemy operacje na ciągach jako operacje na listach, nasze programy w każdym kroku procesu muszą konstruować i kopiować struktury danych (które mogą być ogromne).

Aby zobaczyć, że tak jest faktycznie, porównajmy dwa programy obliczające sumę wszystkich liczb pierwszych z zadanego przedziału. Pierwszy z nich jest napisany w zwykłym stylu iteracyjnym<sup>53</sup>:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

Drugi program wykonuje te same obliczenia, używając operacji na ciągach z punktu 2.2.3:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

Wykonując obliczenia, pierwszy program musi pamiętać tylko kumulowaną sumę. Natomiast w drugim programie filtr nie może niczego sprawdzać, dopóki `enumerate-interval` nie skonstruuje listy wszystkich liczb z danego przedziału. Filtr generuje jeszcze jedną listę, która z kolei, zanim zostanie skrócona do sumy, jest przekazywana do `accumulate`. Taka duża pośrednia struktura danych nie jest potrzebna w pierwszym programie, na który możemy spojrzeć w następujący sposób: liczby należące do przedziału są wyliczane stopniowo i te spośród nich, które są pierwsze, w miarę ich generowania są dodawane do sumy.

---

<sup>53</sup> Zakładamy, że jest dostępny predykat `prime?` (np. taki jak w punkcie 1.2.6) sprawdzający, czy liczba jest pierwsza.

Taka nieefektywność wynikająca z użycia list staje się w widoczny sposób dokuczliwa, gdy zastosujemy paradymat ciągów do wyznaczenia drugiej liczby pierwszej należącej do przedziału od 10 000 do 1 000 000, obliczając wyrażenie

```
(car (cdr (filter prime?
                    (enumerate-interval 10000 1000000))))
```

Wyrażenie to znajduje drugą liczbę pierwszą z przedziału, ale związany z tym narzut obliczeniowy jest niesłychany. Tworzona jest lista złożona z prawie miliona liczb całkowitych, która jest potem przepuszczana przez filtr sprawdzający każdy element, czy jest on liczbą pierwszą, a następnie prawie cały wynik jest ignorowany. Przy bardziej tradycyjnym stylu programowania wyliczanie i filtrowanie przeplatłyby się i zostałyby zatrzymane po znalezieniu drugiej liczby pierwszej.

Korzystanie ze strumieni to sprytna koncepcja umożliwiająca użycie operacji na ciągach bez ponoszenia kosztu związanego z reprezentowaniem ciągów w postaci list. Stosując strumienie, możemy zachować zalety obu podejść — mamy możliwość eleganckiego formułowania programów za pomocą operacji na ciągach przy jednoczesnym zachowaniu wydajności obliczeń przyrostowych. Podstawowy pomysł polega na tym, aby konstruować strumień tylko częściowo i przekazywać taką częściową konstrukcję do programu pochłaniającego jego elementy. Jeśli ten program zechce uzyskać część strumienia, która nie została jeszcze skonstruowana, to strumień automatycznie utworzy wymaganą część siebie, sprawiając wrażenie, że cały jest dostępny. Innymi słowy, chociaż będziemy pisać programy tak, jakby przetwarzane były całe ciągi, jednak implementacja strumieni będzie zrealizowana tak, że zużywanie elementów strumieni będzie się w sposób automatyczny i niezauważalny przeplatać z ich tworzeniem.

Na pierwszy rzut oka strumienie to zwykłe listy, tylko operacje na nich mają inne nazwy. Mamy jeden konstruktor `cons-stream` i dwa selektory `stream-car` i `stream-cdr`, spełniające następujące warunki:

```
(stream-car (cons-stream x y)) = x
(stream-cdr (cons-stream x y)) = y
```

Istnieje wyróżniony obiekt `the-empty-stream` — strumień pusty, który nie może być wynikiem operacji `cons-stream` i który można rozpoznać za pomocą predykatu `stream-null?`<sup>54</sup>. Tak więc możemy tworzyć i wykorzystywać strumienie zupełnie w ten sam sposób, w jaki tworzymy i używamy list, reprezentując za ich pomocą dane ułożone w ciągi. W szczególności możemy

<sup>54</sup> W implementacji MIT `the-empty-stream` jest równy liście pustej '(), a `stream-null?` oznacza to samo co `null?`.

tworzyć strumieniowe odpowiedniki operacji na listach przedstawionych w rozdziale 2, takich jak `list-ref`, `map` i `for-each`<sup>55</sup>:

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1)))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s))))))
```

Procedura `stream-for-each` jest przydatna do wyświetlania strumieni:

```
(define (display-stream s)
  (stream-for-each display-line s))

(define (display-line x)
  (newline)
  (display x))
```

Chcąc, aby implementacja strumieni w sposób automatyczny i niezauważalny przeplatała tworzenie i używanie strumienia, zrobimy to tak, że `cdr` strumienia będzie obliczany w czasie uzyskiwania do niego dostępu za pomocą procedury `stream-cdr`, a nie w czasie tworzenia strumienia za pomocą `cons-stream`. Taka decyzja implementacyjna przypomina nasze rozważania na temat liczb wymiernych w punkcie 2.1.2, gdzie zobaczyliśmy, że możemy zaimplementować liczby wymierne tak, że licznik i mianownik są skracane albo w czasie tworzenia ułamków, albo w czasie wydobywania z nich liczników i mianowników. Obie implementacje liczb wymiernych tworzą tę samą abstrakcję danych, ale różnią się efektywnością. Podobny związek zachodzi między strumieniami i zwykłymi listami. Jako abstrakcje danych strumienie są tym samym co listy. Różnica dotyczy czasu, czyli tego, kiedy elementy są

<sup>55</sup> Powinno nas to zastanowić. Fakt, że definiujemy tak podobne procedury dla strumieni i list, wskazuje, że pominieliśmy kryjącą się za tym jakąś abstrakcję. Niestety, aby ją wykorzystać, będziemy musieli użyć bardziej wyszukanych narzędzi sterujących procesem obliczania niż te, którymi dysponujemy w tej chwili. Powrócimy jeszcze do tej kwestii pod koniec punktu 3.5.4. W podrozdziale 4.2 opracujemy schemat ujednolicenia list i strumieni.

obliczane. W przypadku list zarówno `car`, jak i `cdr` są obliczane w czasie tworzenia. W przypadku strumieni `cdr` jest obliczany w czasie wydobywania informacji.

Nasza implementacja strumieni będzie się opierać na formie specjalnej o nazwie `delay`. Obliczenie (`delay <wyrażenie>`) nie powoduje obliczenia wyrażenia (`wyrażenie`), ale daje w wyniku tzw. *obiekt odroczony* (ang. *delayed object*), który możemy traktować jako „obietnicę” obliczenia (`wyrażenie`) kiedyś w przyszłości. Formie `delay` towarzyszy procedura `force`, której argumentem jest obiekt odroczony i która oblicza jego wartość — zmuszając `delay` do spełnienia obietnicy. Dalej zobaczymy, jak można zaimplementować `delay` i `force`, ale najpierw użyjmy ich do skonstruowania strumieni.

`Cons-stream` to forma specjalna zdefiniowana tak, że

`(cons-stream <a> <b>)`

jest równoważne

`(cons <a> (delay <b>))`

Oznacza to, że będziemy budować strumienie za pomocą par. Jednak zamiast umieszczać wartość pozostały części strumienia w `cdr` pary, będziemy tam umieszczać obietnicę obliczenia pozostały części strumienia, jeśli kiedykolwiek zajdzie taka potrzeba. Możemy więc teraz zdefiniować `stream-car` i `stream-cdr` jako procedury:

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

`Stream-car` wydobywa `car` pary, a `stream-cdr` wydobywa jej `cdr` i oblicza tak uzyskane wyrażenie odroczone w celu otrzymania pozostały części strumienia<sup>56</sup>.

<sup>56</sup> Pomimo że `stream-car` i `stream-cdr` mogą być zdefiniowane jako procedury, `cons-stream` musi być formą specjalną. Gdyby `cons-stream` było procedurą, to zgodnie z naszym modelem obliczeń obliczenie (`cons-stream <a> <b>`) automatycznie powodowałoby obliczenie (`<b>`), czego właśnie chcemy uniknąć. Z tego samego powodu `delay` musi być formą specjalną, choć `force` może być zwykłą procedurą.

## Implementacja strumieni w działaniu

Aby zobaczyć, jak działa implementacja strumieni, przeanalizujemy przedstawione wcześniej „oburzające” obliczenie liczb pierwszych, sformułowane tym razem przy użyciu strumieni:

```
(stream-car
  (stream-cdr
    (stream-filter prime?
      (stream-enumerate-interval 10000 1000000))))
```

Zobaczmy, że w rzeczywistości jest ono efektywne.

Zaczynamy od wywołania procedury `stream-enumerate-interval` z argumentami 10 000 i 1 000 000. `Stream-enumerate-interval` jest strumieniowym odpowiednikiem procedury `enumerate-interval` (z punktu 2.2.3):

```
(define (stream-enumerate-interval low high)
  (if (> low high)
    the-empty-stream
    (cons-stream
      low
      (stream-enumerate-interval (+ low 1) high))))
```

Tak więc wynik `stream-enumerate-interval` jest tworzony przez `cons-stream` i ma postać<sup>57</sup>

```
(cons 10000
  (delay (stream-enumerate-interval 10001 1000000)))
```

Oznacza to, że wynikiem `stream-enumerate-interval` jest strumień reprezentowany jako para, której `car` jest równy 10 000 i której `cdr` to obietnica wyliczenia dalszych elementów z przedziału, jeśli zajdzie taka potrzeba. Strumień ten jest teraz przepuszczany przez filtr wychwytyjący liczby pierwsze, będący strumieniowym odpowiednikiem procedury `filter` (z punktu 2.2.3):

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
    ((pred (stream-car stream))
      (cons-stream (stream-car stream)
        (stream-filter pred
          (stream-cdr stream))))
    (else (stream-filter pred (stream-cdr stream)))))
```

<sup>57</sup> Pokazane tutaj liczby w rzeczywistości nie pojawiają się w wyrażeniu odroczonym. Tym, co faktycznie się pojawia, jest pierwotne wyrażenie, ale w środowisku, w którym zmienne są związane z odpowiednimi liczbami. Na przykład w miejscu 10001 faktycznie występuje `(+ low 1)` ze zmiennej `low` związaną z wartością 10 000.

Stream-filter sprawdza stream-car strumienia (car pary równy 10 000). Ponieważ nie jest to liczba pierwsza, stream-filter sprawdza stream-cdr strumienia wejściowego. Wywołanie stream-cdr wymusza obliczenie odroczonego stream-enumerate-interval, co tym razem daje w wyniku

```
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
```

Stream-filter sprawdza teraz stream-car tego strumienia, czyli 10 001, stwierdza, że to też nie jest liczba pierwsza, wymusza obliczenie kolejnego stream-cdr itd., aż wartością stream-enumerate-interval jest liczba pierwsza 10 007. Wtedy to wynikiem stream-filter, zgodnie z jego definicją, jest

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

co w tym przypadku jest równe

```
(cons 10007
      (delay
        (stream-filter
          prime?
          (cons 10008
                (delay
                  (stream-enumerate-interval 10009
                    1000000))))))
```

Wynik ten jest teraz przekazywany do stream-cdr w pierwotnym wyrażeniu. Wymusza ono obliczenie odroczonej procedury stream-filter, która z kolei wymusza następne obliczenia odroczonej procedury stream-enumerate-interval, dopóki nie znajdzie kolejnej liczby pierwszej, którą jest 10 009. Na koniec, wynik przekazywany do stream-car w naszym pierwotnym wyrażeniu jest równy

```
(cons 10009
      (delay
        (stream-filter
          prime?
          (cons 10010
                (delay
                  (stream-enumerate-interval 10011
                    1000000))))))
```

Wynikiem stream-car jest 10 009 i to kończy obliczenia. Została zbadana pierwszość tylko tylu liczb, ile było potrzebnych do znalezienia drugiej liczby

pierwszej. Podobnie, zostało wygenerowanych tylko tyle liczb należących do przedziału, ile trzeba było przepuścić przez filtr.

Ogólnie mówiąc, możemy myśleć o obliczeniach odroczonych jako o programowaniu „sterowanym zapotrzebowaniami”, dzięki któremu każdy etap przetwarzania strumieni jest uruchamiany tylko na tyle, aby zaspokoić potrzeby następnego etapu. Rozdzieliśmy tutaj faktyczną kolejność zdarzeń w obliczeniu od widocznej struktury procedur. Piszemy procedury tak, jakby strumienie istniały „od razu w całości”, podczas gdy w rzeczywistości obliczenia są wykonywane przyrostowo, jak w tradycyjnym stylu programowania.

### Implementacja delay i force

Pomimo że `delay` i `force` mogą się wydawać tajemniczymi operacjami, ich implementacja jest całkiem prosta. `Delay` musi opakować wyrażenie tak, aby mogło być ono później obliczone na żądanie, co możemy uzyskać, traktując wyrażenie po prostu jako treść procedury. `Delay` może być taką formą specjalną, że

```
(delay (wyrażenie))
```

jest lukrem syntaktycznym równoważnym

```
(lambda () (wyrażenie))
```

`Force` wywołuje po prostu procedurę (bezargumentową), która jest wynikiem `delay`; może więc być zaimplementowane jako następująca procedura:

```
(define (force delayed-object)
  (delayed-object))
```

Ta implementacja wystarcza, żeby `delay` i `force` działały tak, jak to wcześniej opisaliśmy. Jest jednak pewna ważna optymalizacja, którą możemy uwzględnić. W wielu zastosowaniach w rezultacie wielokrotnie wymuszać obliczanie tych samych obiektów odroczonego. W przypadku programów rekurencyjnych używających strumieni może to prowadzić do poważnej nieefektywności. (Zobacz ćwiczenie 3.57). Rozwiążanie polega na takim konstruowaniu obiektów odroczonych, aby zapamiętywały one obliczaną wartość za pierwszym razem, gdy wymuszane jest ich obliczenie. Wynikiem kolejnych wymuszanych obliczeń jest po prostu zapamiętana wartość przekazywana bez powtarzania obliczeń. Innymi słowy, implementujemy `delay` jako wyspecjalizowaną procedurę spamiętującą, podobną do procedury opisanej w ćwiczeniu 3.27. Możemy to zrobić, używając poniższej procedury, której argumentem jest procedura (bezargumentowa), a wynikiem jest wersja tej procedury ze spamiętywaniem. Gdy procedura ze spamiętywaniem jest wywoływana po

raz pierwszy, wtedy zapamiętuje ona obliczony wynik. Kolejne wywołania przekazują po prostu obliczony wynik.

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

Wówczas `delay` jest zdefiniowane tak, że `(delay <wyrażenie>)` jest równoważne

```
(memo-proc (lambda () <wyrażenie>))
```

a procedura `force` jest zdefiniowana tak jak poprzednio<sup>58</sup>.

### Ćwiczenie 3.50

Uzupełnij następujące rozszerzenie definicji `stream-map` na procedury wieloargumentowe, analogicznie do operacji `map` w punkcie 2.2.1, przypis 12.

```
(define (stream-map proc . argstreams)
  (if ((??) (car argstreams))
      the-empty-stream
      ((??)
       (apply proc (map (??) argstreams))
       (apply stream-map
              (cons proc (map (??) argstreams))))))
```

### Ćwiczenie 3.51

Chcąc przyjrzeć się obliczeniom odroczonym, użyjemy następującej procedury, która wypisuje po prostu swój argument i przekazuje go jako wynik:

```
(define (show x)
  (display-line x)
  x)
```

<sup>58</sup> Istnieje wiele możliwości implementacji strumieni, innych niż ta przedstawiona w tym punkcie. Odraczanie obliczeń, które jest kluczem do praktycznej realizacji strumieni, pojawiło się jako nieodłączna część mechanizmu przekazywania argumentów *przez nazwę* w Algolu 60. Użycie tego mechanizmu do zaimplementowania strumieni zostało po raz pierwszy opisane przez Landina w [66]. Odraczanie obliczeń dla strumieni zostało wprowadzone do Lispu przez Friedman i Wise'a [30]. W ich implementacji `cons` zawsze odrcza obliczenie swoich argumentów, a więc automatycznie listy zachowują się jak strumienie. Optymalizacja spamiętywania jest również znana jako obliczanie *na żądanie*. Społeczność algolowa określiłaby nasze pierwotne obiekty odroczone jako *przekazywane przez nazwę* (ang. *call-by-name thunks*), a ich zoptymalizowaną wersję jako *obliczane na żądanie* (ang. *call-by-need thunks*).

Co wypisze interpreter w wyniku obliczenia każdego z następujących wyrażeń<sup>59</sup>?

```
(define x (stream-map show (stream-enumerate-interval 0 10)))
(stream-ref x 5)
(stream-ref x 7)
```

### Ćwiczenie 3.52

Rozważmy następujący ciąg wyrażeń:

```
(define sum 0)
(define (accum x)
  (set! sum (+ x sum))
  sum)

(define seq (stream-map accum (stream-enumerate-interval 1 20)))
(define y (stream-filter even? seq))
(define z (stream-filter (lambda (x) (= (remainder x 5) 0))
                        seq))

(stream-ref y 7)
(display-stream z)
```

Jaka będzie wartość `sum` po obliczeniu każdego z powyższych wyrażeń? Co zostało wypisane w odpowiedzi na obliczenie wyrażeń `stream-ref` i `display-stream`? Czy wyniki te uległyby zmianie, gdyby `(delay (wyrażenie))` było zaimplementowane po prostu jako `(lambda () (wyrażenie))` bez optymalizacji realizowanej przez `memo-proc`? Odpowiedź uzasadnij.

### 3.5.2. Strumienie nieskończone

Zobaczyliśmy, jak sprawić wrażenie, że operujemy na strumieniach jako na całościach, podczas gdy faktycznie jest obliczana tylko taka część strumienia, do jakiej potrzebujemy dostępu. Możemy używać tej techniki do efektywnego reprezentowania nawet bardzo długich ciągów jako strumieni. Bardziej zadziwiające jest to, że możemy stosować strumienie do reprezentowania ciągów nieskończonych. Rozważmy na przykład następującą definicję strumienia do datnych liczb całkowitych:

---

<sup>59</sup> Takie ćwiczenia, jak 3.51 i 3.52, mają za zadanie sprawdzenie naszego zrozumienia tego, jak działa `delay`. Jednakże mieszanie ze sobą obliczeń odroczonych i wypisywania — lub, jeszcze gorzej, przypisań — jest niezwykle mylące, a prowadzący zajęcia z języków programowania tradycyjnie torturują swoich studentów takimi zadaniami egzaminacyjnymi, jak te w niniejszym punkcie. Nie mówiąc już o tym, że pisanie programów, które zależą od takich subtelności, świadczy o tragicznym stylu programowania. Część siły przetwarzania strumieni polega na tym, że możemy nie przejmować się kolejnością, w jakiej zdarzenia faktycznie zchodzą w programach. Niestety, jest to coś, na co akurat nie możemy sobie pozwolić, używając przypisań, gdyż zmuszają nas one do uwzględnienia czasu i zmiany.

---

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1)))))

(define integers (integers-starting-from 1))
```

Definicja taka ma sens, gdyż `integers` jest parą, której car jest równy 1, a cdr stanowi obietnicę wyliczenia liczb całkowitych od 2 w górę. Jest to strumień nieskończony, ale w każdej chwili możemy zbadać tylko jego skończoną część. Tak więc nasze programy nigdy nie zauważą, że strumień nie istnieje w całości.

Za pomocą `integers` możemy zdefiniować inne strumienie nieskończone, takie jak strumień liczb całkowitych niepodzielnych przez 7:

```
(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
    integers))
```

Wówczas możemy wyznaczać liczby całkowite niepodzielne przez 7, odwołując się po prostu do elementów strumienia:

```
(stream-ref no-sevens 100)
117
```

Analogicznie do `integers` możemy zdefiniować nieskończony strumień liczb Fibonacciego:

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b)))))

(define fibs (fibgen 0 1))
```

`Fibs` to para, której car jest równy 0, a cdr jest obietnicą obliczenia (`fibgen 1 1`). Gdy obliczamy odroczone (`fibgen 1 1`), powstaje para, której car jest równy 1 i której cdr jest obietnicą obliczenia (`fibgen 1 2`) itd.

Aby przyjrzeć się bardziej interesującemu strumieniowi nieskończonemu, możemy uogólnić przykładowy strumień `no-sevens` i stosując metodę znaną jako *sito Eratostenesa*, zbudować nieskończony strumień liczb pierwszych<sup>60</sup>. Zaczynamy od liczb całkowitych, od 2 (która jest najmniejszą liczbą pierwszą) w górę. Chcąc otrzymać pozostałe liczby pierwsze, odsiewamy (filtrując)

---

<sup>60</sup> Eratostenes — grecki filozof żyjący w Aleksandrii w III w. p.n.e. Znany jest z tego, że podał pierwsze dokładne oszacowanie obwodu Ziemi, który wyznaczył, obserwując cienie rzucane w południe w dzień przesilenia letniego. Metoda sita Eratostenesa, pomimo że starożytna, była podstawą budowy wyspecjalizowanych sprzętowych „sit”, które jeszcze do niedawna były najpotężniejszym istniejącym narzędziem służącym do wyznaczania dużych liczb pierwszych. Jednak począwszy od lat siedemdziesiątych XX w., metody te zostały wyparte przez bujnie rozwijające się techniki probabilistyczne, które omówiliśmy w punkcie 1.2.6.

jemy) wielokrotności 2. Pozostaje nam strumień zaczynający się od 3, która jest kolejną liczbą pierwszą. Z pozostałych liczb całkowitych odsiewamy teraz wielokrotności 3. Pozostaje nam strumień zaczynający się od 5, która jest kolejną liczbą pierwszą, itd. Innymi słowy, konstruujemy liczby pierwsze za pomocą następującego procesu przesiewania: Aby przesieć strumień S, tworzymy strumień, którego pierwszy element jest pierwszym elementem S, a jego pozostałą część uzyskujemy, odsiewając wielokrotności pierwszego elementu S z pozostałą częścią S i przesiewając dalej tak uzyskany wynik. Proces ten można łatwo opisać za pomocą operacji na strumieniach:

```
(define (sieve stream)
  (cons-stream
    (stream-car stream)
    (sieve (stream-filter
      (lambda (x)
        (not (divisible? x (stream-car stream)))))
      (stream-cdr stream)))))

(define primes (sieve (integers-starting-from 2)))
```

Teraz, chcąc wyznaczyć daną liczbę pierwszą, wystarczy, że o nią poprosimy:

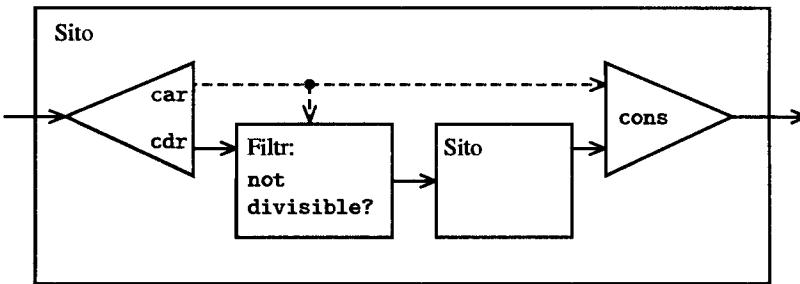
```
(stream-ref primes 50)
233
```

Ciekawe może być rozważenie systemu przetwarzania sygnałów, jaki tworzy `sieve`, przedstawionego na „diagramie Hendersona” pokazanym na rys. 3.31<sup>61</sup>. Strumień wejściowy jest przepuszczany przez „dekonstraktor”, który oddziela pierwszy element strumienia od jego pozostałej części. Pierwszy element jest użyty do budowy filtru podzielności, przez który jest przepuszczana pozostała część strumienia, a po przefiltrowaniu jest przekazywana do kolejnego sita. Następnie z pierwszego elementu strumienia początkowego oraz strumienia wychodzącego z wewnętrznego sita jest konstruowany strumień wyjściowy. Tak więc nie tylko strumień jest nieskończony, ale i sam przetwornik sygnałów jest nieskończony, gdyż każde sito zawiera w środku kolejne sito.

### Uwikłane definicje strumieni

Strumienie `integers` i `fibs` były wcześniej zdefiniowane przez określenie procedur „generujących”, które bezpośrednio obliczają elementy strumienia

<sup>61</sup> Nazwaliśmy ten diagram od nazwiska Petera Hendersona, który pierwszy pokazał nam tego typu diagramy jako sposób przedstawiania przetwarzania strumieni. Każda linia ciągła reprezentuje przekazywany strumień wartości. Linie przerywane, od `car` do `cons` i `filter`, wskazują, że jest przekazywana pojedyncza wartość, a nie strumień.



Rys. 3.31. Przedstawienie sita liczb pierwszych jako systemu przetwarzania sygnałów

jeden po drugim. Alternatywna metoda określania strumieni polega na użyciu obliczeń odroczonych do definiowania strumieni w sposób uwikłany. Następujące wyrażenie definiuje na przykład strumień ones jako nieskończony strumień jedynek:

```
(define ones (cons-stream 1 ones))
```

Definicja ta działa podobnie do definicji procedur rekurencyjnych — ones jest parą, której car jest równy 1, a cdr jest obietnicą obliczenia ones. Wynikiem obliczenia cdr jest znowu 1 i obietnica obliczenia ones itd.

Możemy robić ciekawsze rzeczy, operując na strumieniach za pomocą operacji takich jak `add-streams`, której wynikiem jest strumień złożony z sum kolejnych elementów dwóch strumieni<sup>62</sup>:

```
(define (add-streams s1 s2)
  (stream-map + s1 s2))
```

Możemy teraz zdefiniować strumień liczb całkowitych w następujący sposób:

```
(define integers (cons-stream 1 (add-streams ones integers)))
```

Definicja ta określa `integers` jako strumień, którego pierwszy element jest równy 1, a pozostała część składa się z sum kolejnych elementów strumieni `ones` i `integers`. Tak więc drugi element `integers` jest równy pierwszemu elementowi `integers` plus 1, czyli 2; trzeci element `integers` jest równy drugiemu elementowi plus 1, czyli 3; itd. Definicja ta działa, gdyż w każdym momencie wystarczająca część strumienia `integers` jest już wygenerowana i może być ponownie użyta do wyznaczenia kolejnej liczby całkowitej.

W podobny sposób możemy zdefiniować liczby Fibonacciego:

```
(define fibs
  (cons-stream 0
```

<sup>62</sup> Użyta tu jest uogólniona wersja `stream-map` z ćwiczenia 3.50.

---

```
(cons-stream 1
            (add-streams (stream-cdr fibs)
                         fibs))))
```

Definicja ta określa, że **fibs** jest takim strumieniem, który zaczyna się od 0 i 1, a pozostała jego część może być wygenerowana przez dodanie elementów **fibs** do elementów tego samego strumienia, ale przesuniętych o jedną pozycję:

1	1	2	3	5	8	13	21	...	= (stream-cdr fibs)
0	1	1	2	3	5	8	13	...	= fibs
0 1 1 2 3 5 8 13 21 34 ... = fibs									

**Scale-stream** to inna procedura przydatna przy formułowaniu takich definicji strumieni. Mnoży ona każdy element strumienia przez daną stałą:

```
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor)) stream))
```

Na przykład wynikiem procedury

```
(define double (cons-stream 1 (scale-stream double 2)))
```

jest strumień złożony z kolejnych potęg dwójki: 1, 2, 4, 8, 16, 32, ...

Możemy sformułować alternatywną definicję strumienia liczb pierwszych, biorąc strumień liczb całkowitych i przepuszczając go przez filtr pozostawiający tylko liczby pierwsze. Na początek będzie nam potrzebna najmniejsza liczba pierwsza, czyli 2:

```
(define primes
  (cons-stream
    2
    (stream-filter prime? (integers-starting-from 3))))
```

Definicja ta nie jest tak prosta, jakby się to wydawało, gdyż będziemy badać, czy  $n$  jest liczbą pierwszą, sprawdzając, czy jest ona podzielna przez jakąś liczbę pierwszą (a nie po prostu przez liczbę całkowitą) mniejszą lub równą  $\sqrt{n}$ :

```
(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) true)
          ((divisible? n (stream-car ps)) false)
          (else (iter (stream-cdr ps))))))
  (iter primes))
```

Jest to definicja rekurencyjna, gdyż **primes** zostało zdefiniowane za pomocą predykatu **prime?**, który z kolei używa strumienia **primes**. Definicja ta dzia-

ła, gdyż w każdym momencie jest wygenerowana dostateczna część strumienia `primes`, aby móc sprawdzać, czy kolejne liczby całkowite są liczbami pierwszymi. Oznacza to, że dla każdego  $n$  sprawdzamy, czy jest to liczba pierwsza — albo  $n$  nie jest pierwsza (wówczas istnieje wygenerowana już liczba pierwsza, która jest dzielnikiem  $n$ ), albo  $n$  jest liczbą pierwszą (wówczas istnieje wygenerowana już liczba pierwsza — tzn. liczba pierwsza mniejsza od  $n$  — która jest większa od  $\sqrt{n}$ )<sup>63</sup>.

### Ćwiczenie 3.53

Nie uruchamiając programu, opisz elementy strumienia zdefiniowanego w następujący sposób:

```
(define s (cons-stream 1 (add-streams s s)))
```

### Ćwiczenie 3.54

Zdefiniuj procedurę `mul-streams`, analogiczną do procedury `add-streams`, której wynikiem jest strumień iloczynów kolejnych elementów dwóch danych strumieni. Użyj tej procedury oraz strumienia `integers` do uzupełnienia następującej definicji strumienia, którego  $n$ -ty element (numerując elementy od 0) jest równy  $(n + 1)!$ :

```
(define factorials (cons-stream 1 (mul-streams (??) (??))))
```

### Ćwiczenie 3.55

Zdefiniuj procedurę `partial-sums`, której argumentem jest strumień  $S$ , a wynikiem jest strumień o elementach równych:  $S_0, S_0 + S_1, S_0 + S_1 + S_2, \dots$ . Na przykład (`partial-sums integers`) powinno być strumieniem  $1, 3, 6, 10, 15, \dots$

### Ćwiczenie 3.56

Stłynny problem, sformułowany po raz pierwszy przez R. Hamminga, polega na wyliczeniu w rosnącej kolejności, bez powtórzeń, wszystkich liczb całkowitych, których rozkłady na iloczyny liczb pierwszych zawierają jedynie liczby 2, 3 i 5. Oczwiste rozwiążanie tego problemu polega po prostu na sprawdzaniu, czy w rozkładach kolejnych liczb całkowitych występują czynniki różne od 2, 3 i 5. Jest to jednak rozwiązanie bardzo nieefektywne, gdyż dla dużych liczb całkowitych coraz mniej z nich spełnia podane wymagania. Spróbujmy spojrzeć na ten problem inaczej. Oznaczmy szukany strumień liczb przez  $S$ . Zauważmy, że ma on następujące własności:

- pierwszym elementem  $S$  jest 1;
- elementy strumienia (`scale-stream S 2`) są również elementami strumienia  $S$ ;

<sup>63</sup> Ten ostatni punkt jest bardzo subtelny i opiera się na fakcie, że  $p_{n+1} \leq p_n^2$ , gdzie  $p_k$  oznacza  $k$ -tą liczbę pierwszą. Tego typu oszacowania są bardzo trudne do uzyskania. Starożytny dowód Euklidesa na istnienie nieskończoności wielu liczb pierwszych pokazuje, że  $p_{n+1} \leq p_1 p_2 \cdots p_n + 1$  i żaden istotnie lepszy rezultat nie był znany aż do 1851 r., gdy rosyjski matematyk P. L. Cebyszew udowodnił, że dla każdego  $n$  zachodzi  $p_{n+1} \leq 2p_n$ . Wynik ten, pierwotnie sformułowany w 1845 r., jest znany jako *hipoteza Bertranda*. Jego dowód można znaleźć w książce Hardy'ego i Wrighta [44], p. 22.3.

- tak samo jest dla `(scale-stream S 3)` i `(scale-stream S 5)`;
- żadne inne elementy nie należą do S.

Musimy teraz jedynie połączyć elementy pochodzące z tych źródeł. W tym celu zdefiniujemy procedurę `merge`, która scalą dwa uporządkowane strumienie w jeden uporządkowany strumień, usuwając przy tym powtórzenia:

```
(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
          (let ((s1car (stream-car s1))
                (s2car (stream-car s2)))
            (cond ((< s1car s2car)
                  (cons-stream s1car (merge (stream-cdr s1) s2)))
                  ((> s1car s2car)
                  (cons-stream s2car (merge s1 (stream-cdr s2))))
                  (else
                    (cons-stream s1car
                      (merge (stream-cdr s1)
                            (stream-cdr s2))))))))
```

Wówczas za pomocą `merge` możemy skonstruować szukany strumień w następujący sposób:

```
(define S (cons-stream 1 (merge (??) (??))))
```

Wstaw brakujące wyrażenia w miejsca oznaczone `(??)`.

### Ćwiczenie 3.57

Ille dodawań jest wykonywanych, gdy obliczamy  $n$ -tą liczbę Fibonacciego, używając strumienia `fibs` zdefiniowanego za pomocą procedury `add-streams?` Pokaż, że liczba dodawań byłaby wykładniczo większa, gdyby `(delay (wyrażenie))` było zaimplementowane tylko jako `(lambda () (wyrażenie))`, bez optymalizacji wykonywanej przez procedurę `memo-proc` (opisaną w punkcie 3.5.1)<sup>64</sup>.

### Ćwiczenie 3.58

Podaj interpretację strumienia obliczanego przez następującą procedurę:

```
(define (expand num den radix)
  (cons-stream
    (quotient (* num radix) den)
    (expand (remainder (* num radix) den) den radix)))
```

<sup>64</sup> Ćwiczenie to pokazuje, jak ściśle obliczanie na żądanie jest związane ze zwykłym spamiętywaniem opisany w ćwiczeniu 3.27. W ćwiczeniu tym użyliśmy przypisania do jawnej konstrukcji tablicy lokalnej. Optymalizacja obliczania na żądanie faktycznie tworzy taką tablicę automatycznie, przechowując wartości w obliczonych już częściach strumienia.

(Quotient to procedura pierwotna, której wynikiem jest całkowita część ilorazu dwóch liczb całkowitych). Jakie są kolejne elementy strumienia (expand 1 7 10) ? Co jest wynikiem (expand 3 8 10) ?

### Ćwiczenie 3.59

W punkcie 2.5.3 widzieliśmy, jak można zaimplementować arytmetykę wielomianów, reprezentując wielomiany w postaci list wyrazów. W podobny sposób możemy操作 szeregi potęgowymi, takimi jak

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \dots$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \dots$$

$$\sin x = x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \dots$$

reprezentowanymi jako strumienie nieskończone. Szereg  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  będziemy reprezentować w postaci strumienia, którego elementy są współczynnikami  $a_0, a_1, a_2, a_3, \dots$ .

(a) Całka szeregu  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  jest szeregiem postaci

$$c + a_0x + \frac{1}{2}a_1x^2 + \frac{1}{3}a_2x^3 + \frac{1}{4}a_3x^4 + \dots$$

gdzie  $c$  jest dowolną stałą. Zdefiniuj procedurę `integrate-series`, której argumentem jest strumień  $a_0, a_1, a_2, \dots$  reprezentujący szereg potęgowy, a wynikiem jest strumień  $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \dots$  współczynników zmiennych wyrazów całki danego szeregu. (Ponieważ wynik nie zawiera wyrazu stałego, nie reprezentuje więc szeregu potęgowego — używając `integrate-series` dołączamy odpowiednią stałą za pomocą `cons`).

(b) Funkcja  $x \mapsto e^x$  jest swoją własną całką. Oznacza to, że  $e^x$  i całka z  $e^x$  mają, z dokładnością do wyrazu stałego  $e^0 = 1$ , takie same szeregi potęgowe. W związku z tym możemy w następujący sposób wygenerować szereg potęgowy  $e^x$ :

```
(define exp-series
  (cons-stream 1 (integrate-series exp-series)))
```

Pokaż, jak można wygenerować szeregi potęgowe dla funkcji sinus i cosinus, wykorzystując fakt, że pochodną sinusa jest cosinus, a pochodną cosinusa jest minus sinus:

```
(define cosine-series
  (cons-stream 1 (??)))
(define sine-series
  (cons-stream 0 (??)))
```

### Ćwiczenie 3.60

Przy reprezentacji szeregów potęgowych w postaci strumieni współczynników, jak w ćwiczeniu 3.59, dodawanie szeregów jest zaimplementowane przez `add-streams`.

Uzupełnij następującą procedurę mnożenia szeregów:

```
(define (mul-series s1 s2)
  (cons-stream (??) (add-streams (??) (??))))
```

Możesz przetestować swoją procedurę, sprawdzając za pomocą szeregów z ćwiczenia 3.59, że  $\sin^2 x + \cos^2 x = 1$ .

### Ćwiczenie 3.61

Niech  $S$  będzie szeregiem potęgowym (ćwiczenie 3.59), którego wyraz stały jest równy 1. Przypuśćmy, że chcemy wyznaczyć szereg potęgowy  $1/S$ , tzn. taki szereg  $X$ , że  $S \cdot X = 1$ . Niech  $S = 1 + S_R$ , gdzie  $S_R$  jest częścią szeregu  $S$  występującą po wyrazie stałym. Wówczas możemy wyznaczyć  $X$  w następujący sposób:

$$\begin{aligned} S \cdot X &= 1 \\ (1 + S_R) \cdot X &= 1 \\ X + S_R \cdot X &= 1 \\ X &= 1 - S_R \cdot X \end{aligned}$$

Innymi słowy,  $X$  jest szeregiem potęgowym, którego wyraz stały jest równy 1, a dalsze wyrazy są przeciwe do wyrazów  $S_R$  razy  $X$ . Użyj tego pomysłu do napisania procedury `invert-unit-series`, która dla szeregu potęgowego  $S$  o wyrazie stałym równym 1 oblicza  $1/S$ . Potrzebna Ci będzie procedura `mul-series` z ćwiczenia 3.60.

### Ćwiczenie 3.62

Zastosuj wyniki ćwiczeń 3.60 i 3.61 do zdefiniowania procedury `div-series` dzielącej przez siebie dwa szeregi potęgowe. `Div-series` powinna działać dla dowolnych dwóch szeregów, pod warunkiem że szereg w mianowniku zaczyna się od niezerowego wyrazu stałego. (Jeśli mianownik ma zerowy wyraz stały, procedura `div-series` powinna zgłaszać błąd). Pokaż, jak za pomocą `div-series` oraz wyników ćwiczenia 3.59 można wygenerować szereg potęgowy funkcji tangens.

### 3.5.3. Zastosowanie paradygmatu strumieniowego

Strumienie z obliczaniem odroczonym mogą stanowić potężne narzędzie modelowania, oferując wiele spośród korzyści, jakie niosą ze sobą stany lokalne i przypisanie. Ponadto pozwalają uniknąć niektórych płatów trudnych pojęć teoretycznych, jakie towarzyszą wprowadzeniu przypisania do języka programowania.

Podejście strumieniowe może być pouczające, gdyż umożliwia budowanie systemów o innych granicach między modułami niż w systemach zorganizowanych wokół przypisów i zmiennych stanowych. Możemy na przykład skupić się na całym szeregu czasowym (lub sygnale), a nie na wartościach zmiennych stanu w poszczególnych momentach czasu. Dzięki temu łatwiej można łączyć i porównywać składowe stanów z różnych momentów czasu.

## Sformułowanie iteracji w postaci przetwarzania strumieni

W punkcie 1.2.1 wprowadziliśmy pojęcie procesu iteracyjnego, którego działanie polega na wielokrotnym modyfikowaniu zmiennych stanu. Wiemy już, że możemy reprezentować stan w postaci „ponadczasowego” strumienia wartości, a nie w postaci zestawu modyfikowanych zmiennych. Przyjmijmy taki punkt widzenia i wróćmy do procedury pierwiastkowania, przedstawionej w punkcie 1.1.7. Przypomnijmy, że pomysł polegał na generowaniu ciągu coraz lepszych przybliżeń pierwiastka kwadratowego z  $x$  za pomocą wielokrotnego stosowania procedury polepszającej przybliżenia:

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
```

W oryginalnej procedurze `sqrt` modelowaliśmy kolejne przybliżenia za pomocą kolejnych wartości zmiennej stanu. Zamiast tego możemy generować nieskończony strumień przybliżeń, zaczynając od początkowego przybliżenia równego 1<sup>65</sup>:

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
      (stream-map (lambda (guess)
                    (sqrt-improve guess x))
                  guesses)))
  guesses)

(display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
...
```

Możemy generować coraz więcej elementów strumienia, otrzymując coraz lepsze przybliżenia. Jeśli chcemy, możemy też napisać procedurę, która generuje kolejne elementy aż do uzyskania dostatecznie dobrego przybliżenia. (Zobacz ćwiczenie 3.64).

Inna iteracja, którą możemy potraktować w ten sam sposób, polega na przybliżaniu  $\pi$  za pomocą szeregu naprzemiennego przedstawionego w punkcie 1.3.1:

<sup>65</sup> Nie możemy użyć tutaj `let` do związania zmiennej lokalnej `guesses`, ponieważ wartość `guesses` zależy od niej samej. W ćwiczeniu 3.63 zajmujemy się tym, dlaczego potrzebna jest nam tutaj zmienna lokalna.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

Najpierw generujemy strumień składników szeregu (odwrotności nieparzystych liczb całkowitych, o naprzemiennych znakach). Następnie tworzymy strumień coraz dłuższych sum częściowych (za pomocą procedury `partial-sums` z ćwiczenia 3.55) i mnożymy wynik przez 4:

```
(define (pi-summands n)
  (cons-stream (/ 1.0 n)
              (stream-map - (pi-summands (+ n 2)))))

(define pi-stream
  (scale-stream (partial-sums (pi-summands 1)) 4))

(display-stream pi-stream)
4.
2.666666666666667
3.466666666666667
2.8952380952380956
3.3396825396825403
2.9760461760461765
3.2837384837384844
3.017071817071818
...
```

W ten sposób uzyskujemy strumień (ciąg) coraz lepszych przybliżeń  $\pi$ , chociaż zbiega on raczej pomału. Osiem wyrazów ciągu ogranicza wartość  $\pi$  do przedziału od 3.284 do 3.017.

Jak dotąd nasze podejście polegające na zastosowaniu strumienia stanów nie różniło się bardzo od modyfikowania zmiennych stanu. Jednak strumienie umożliwiają nam wykonanie kilku ciekawych sztuczek. Możemy na przykład przekształcić strumień za pomocą *akceleratora ciągów* zmieniającego ciąg przybliżeń w nowy ciąg, który zbiega do tej samej wartości, ale szybciej.

Jeden z takich akceleratorów, autorstwa XVIII-wiecznego szwajcarskiego matematyka Leonharda Eulera, działa dobrze dla ciągów, które są sumami częściowymi szeregów naprzemiennych (tzn. szeregów, których wyrazy mają naprzemienne znaki). W metodzie Eulera, jeśli  $S_n$  jest  $n$ -tym wyrazem danego ciągu sum częściowych, to wynikiem akceleracji jest ciąg o wyrazach

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Tak więc, jeśli pierwotny ciąg jest reprezentowany w postaci strumienia wartości, to przekształcony ciąg jest określony przez

---

```
(define (euler-transform s)
  (let ((s0 (stream-ref s 0)) ;  $S_{n-1}$ 
        (s1 (stream-ref s 1)) ;  $S_n$ 
        (s2 (stream-ref s 2))) ;  $S_{n+1}$ 
    (cons-stream (- s2 (/ (square (- s2 s1))
                           (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))
```

Działanie akceleratora Eulera możemy pokazać na przykładzie naszego ciągu przybliżeń  $\pi$ :

```
(display-stream (euler-transform pi-stream))
3.1666666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

Jeszcze lepiej, możemy akcelerować zakcelerowany ciąg, następnie rekurencyjnie go jeszcze zakcelerować itd. Tworzymy mianowicie strumień strumieni (strukturę, którą będziemy nazywać *tableau*), w której każdy strumień jest wynikiem przekształcenia poprzedniego strumienia:

```
(define (make-tableau transform s)
  (cons-stream s
    (make-tableau transform
      (transform s))))
```

Tableau ma postać

$s_{00}$	$s_{01}$	$s_{02}$	$s_{03}$	$s_{04}$	...
$s_{10}$	$s_{11}$	$s_{12}$	$s_{13}$	...	
$s_{20}$	$s_{21}$	$s_{22}$	...		
					...

Na koniec tworzymy ciąg złożony z pierwszych wyrazów kolejnych wierszy tableau:

```
(define (accelerated-sequence transform s)
  (stream-map stream-car
    (make-tableau transform s)))
```

Możemy pokazać działanie takiej „superakceleracji” na przykładzie ciągu zbieżnego do  $\pi$ :

```
(display-stream (accelerated-sequence euler-transform
                                         pi-stream))
```

```
4.  
3.1666666666666667  
3.142105263157895  
3.141599357319005  
3.1415927140337785  
3.1415926539752927  
3.1415926535911765  
3.141592653589778
```

...

Wynik jest imponujący. Obliczenie pierwszych ośmiu wyrazów ciągu daje nam przybliżenie poprawnej wartości  $\pi$  z dokładnością do 14 miejsc dziesiętnych. Gdybyśmy użyli tylko pierwotnego ciągu zbieżnego do  $\pi$ , to żeby uzyskać taką samą dokładność, musielibyśmy obliczyć rzędu  $10^{13}$  wyrazów (tzn. rozwinać szereg do miejsca, w którym poszczególne wyrazy są mniejsze niż  $10^{-13}$ ).

Moglibyśmy zaimplementować taką technikę akceleracji bez używania strumieni. Jednakże sformułowanie strumieniowe jest szczególnie eleganckie i poręczne, gdyż cały ciąg stanów jest dostępny w postaci struktury danych, na której możemy operować za pomocą jednolitego zestawu operacji.

### Ćwiczenie 3.63

Ludwik Myślicielak pyta się, dlaczego procedura `sqrt-stream` nie została zapisana w następujący, prostszy sposób bez zmiennej lokalnej `guesses`:

```
(define (sqrt-stream x)
  (cons-stream 1.0
               (stream-map (lambda (guess)
                             (sqrt-improve guess x))
                           (sqrt-stream x))))
```

Liz P. Haker odpowiada mu, że taka wersja procedury jest znacznie mniej efektywna, gdyż wykonuje nadmiarowe obliczenia. Wyjaśnij odpowiedź Liz. Czy obie wersje różniłyby się efektywnością, gdyby nasza implementacja `delay` sprowadzała się tylko do `(lambda () (wyrażenie))`, bez optymalizacji realizowanej przez procedurę `memo-proc` (punkt 3.5.1)?

### Ćwiczenie 3.64

Napisz procedurę `stream-limit`, której argumentami są strumień i liczba (tolerancja). Powinna ona badać strumień aż do napotkania dwóch kolejnych elementów, które różnią się co do wartości bezwzględnej o mniej niż zadana tolerancja, i przekazywać

drugi z tych elementów jako swój wynik. Za pomocą takiej procedury moglibyśmy obliczyć pierwiastek kwadratowy z zadaną dokładnością w następujący sposób:

```
(define (sqrt x tolerance)
  (stream-limit (sqrt-stream x) tolerance))
```

### Ćwiczenie 3.65

Użyj szeregu

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

do skonstruowania trzech ciągów zbieżnych do logarytmu naturalnego 2, w ten sam sposób, jak robiliśmy to dla liczby  $\pi$ . Jak szybko te ciągi zbiegają do granicy?

### Nieskończone strumienie par

W punkcie 2.2.3 widzieliśmy, jak zgodnie z paradygmatem ciągowym można ująć tradycyjne pętle zagnieździane jako procesy przetwarzające ciągi par. Jeśli uogólnimy tę technikę na strumienie nieskończone, to będziemy mogli pisać programy, które już nie tak łatwo reprezentują się przy użyciu pętli, gdyż pętle muszą przebiegać zbiory nieskończone.

Przypuśćmy na przykład, że chcemy uogólnić procedurę `prime-sum-pairs` z punktu 2.2.3 tak, aby jej wynikiem był strumień par *wszystkich* takich liczb całkowitych  $(i, j)$ , że  $i \leq j$  oraz  $i + j$  jest liczbą pierwszą. Jeśli `int-pairs` jest ciągiem wszystkich par liczb całkowitych  $(i, j)$ , przy czym  $i \leq j$ , to nasz szukany strumień to po prostu<sup>66</sup>:

```
(stream-filter (lambda (pair)
  (prime? (+ (car pair) (cadr pair))))
  int-pairs)
```

Tak więc nasz problem sprowadza się do utworzenia strumienia `int-pairs`. Ogólniej, przypuśćmy, że mamy dwa strumienie  $S = (S_i)$  i  $T = (T_j)$ , i wyobraźmy sobie nieskończoną tablicę prostokątną

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
$(S_1, T_0)$	$(S_1, T_1)$	$(S_1, T_2)$	$\dots$
$(S_2, T_0)$	$(S_2, T_1)$	$(S_2, T_2)$	$\dots$
$\dots$			

<sup>66</sup> Tak jak w punkcie 2.2.3, reprezentujemy tu pary liczb całkowitych jako listy, a nie jako pary lispowe.

Chcielibyśmy wygenerować strumień zawierający wszystkie pary znajdujące się w tablicy na przekątnej lub powyżej niej, tzn. pary

$$\begin{array}{cccc}
 (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \dots \\
 & (S_1, T_1) & (S_1, T_2) & \dots \\
 & & (S_2, T_2) & \dots \\
 & & & \dots
 \end{array}$$

(Jeśli przyjmiemy, że zarówno  $S$ , jak i  $T$  są strumieniami liczb całkowitych, to wynikiem będzie szukany strumień `int-pairs`).

Oznaczmy ogólny strumień takich par przez `(pairs S T)` i przyjmijmy, że składa się on z trzech części: pary  $(S_0, T_0)$ , pozostałych par z pierwszego wiersza oraz reszty par<sup>67</sup>:

$$\begin{array}{c|cccc}
 (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \dots \\
 \hline
 & (S_1, T_1) & (S_1, T_2) & \dots \\
 & & (S_2, T_2) & \dots \\
 & & & \dots
 \end{array}$$

Zauważmy, że przy takim podziale trzecia składowa (pary, które nie znajdują się w pierwszym wierszu) jest zbudowana (rekurencyjnie) z par elementów strumieni `(stream-cdr S)` i `(stream-cdr T)`. Zwróćmy również uwagę, że druga składowa (pozostała część pierwszego wiersza) to

```
(stream-map (lambda (x) (list (stream-car s) x))
            (stream-cdr t))
```

A zatem możemy zbudować strumień par w następujący sposób:

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    ((połącz w jakiś sposób)
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (pairs (stream-cdr s) (stream-cdr t)))))
```

Aby uzupełnić procedurę, musimy wybrać jakiś sposób połączenia dwóch wewnętrznych strumieni. Jeden z pomysłów polega na użyciu strumieniowego odpowiednika procedury `append` z punktu 2.2.1:

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
    s2
```

<sup>67</sup> Zobacz ćwiczenie 3.68, żeby zrozumieć, dlaczego wybraliśmy taki podział.

---

```
(cons-stream (stream-car s1)
             (stream-append (stream-cdr s1) s2))))
```

Procedura ta nie nadaje się jednak do łączenia strumieni nieskończonych, gdyż elementy pierwszego strumienia poprzedzają elementy drugiego strumienia. W szczególności, gdybyśmy spróbowali wygenerować wszystkie pary dodatkowych liczb całkowitych za pomocą

```
(pairs integers integers)
```

powstały strumień starałby się wyliczyć najpierw wszystkie pary, których pierwsza składowa jest równa 1, i w rezultacie nigdy nie wygenerowałby par zawierających w pierwszej składowej jakiekolwiek innej liczby.

Chcąc obsługiwać strumienie nieskończone, musimy wymyślić taką kolejność łączenia, która gwarantuje, że każdy element zostanie w końcu osiągnięty, o ile nasz program będzie działał dostatecznie długo. Następująca procedura `interleave` stanowi elegancki sposób osiągnięcia tego celu<sup>68</sup>.

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

Ponieważ `interleave` pobiera elementy na przemian z obydwu strumieni, każdy element drugiego strumienia zostanie w końcu umieszczony w strumieniu wynikowym, nawet jeśli pierwszy strumień jest nieskończony.

Możemy więc wygenerować potrzebny strumień par w następujący sposób:

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (pairs (stream-cdr s) (stream-cdr t)))))
```

### Ćwiczenie 3.66

Zbadaj strumień `(pairs integers integers)`. Czy potrafisz ogólnie opisać kolejność, w jakiej pary pojawiają się w tym strumieniu? Na przykład, ile w przybliżeniu

---

<sup>68</sup> Dokładne sformułowanie własności, jakiej wymagamy od kolejności złożenia, jest następujące: powinna istnieć taka dwuargumentowa funkcja  $f$ , że para złożona z  $i$ -tego elementu pierwszego strumienia i  $j$ -tego elementu drugiego strumienia pojawia się jako  $f(i, j)$ -ta para w strumieniu wynikowym. [Ponadto  $f$  powinna być funkcją wzajemnie jednoznaczna; przyp. tłum.]. Sztuczkę polegającą na uzyskaniu tego efektu za pomocą procedury `interleave` pokazał nam David Turner, który zastosował ją w języku KRC [105].

par poprzedza pary (1,100), (99,100) i (100,100)? (Jeśli potrafisz podać dokładny wzór matematyczny, tym lepiej. Jeśli jednak zaczniesz grzędzać w rachunkach, to wystarczą odpowiedzi szacunkowe).

### Ćwiczenie 3.67

Zmodyfikuj procedurę `pairs` tak, aby wynikiem (`pairs integers integers`) był strumień złożony ze *wszystkich* par dodatnich liczb całkowitych  $(i, j)$  (bez dodatkowego warunku  $i \leq j$ ). Wskazówka: powinieneś wprowadzić dodatkowy strumień.

### Ćwiczenie 3.68

Ludwik Myśliścielak uważa, że budowanie strumienia par z trzech części jest niepotrzebnie skomplikowane. Zamiast oddzielać parę  $(S_0, T_0)$  od reszty par z pierwszego wiersza, proponuje on, aby za jednym zamachem podać cały pierwszy wiersz w następujący sposób:

```
(define (pairs s t)
  (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                t)
    (pairs (stream-cdr s) (stream-cdr t))))
```

Czy to zadziała? Rozważ, co się stanie, jeśli obliczymy (`pairs integers integers`), korzystając z definicji procedury `pairs` zaproponowanej przez Ludwika.

### Ćwiczenie 3.69

Napisz procedurę `triples`, której argumentami są trzy strumienie nieskończone  $S$ ,  $T$  i  $U$  i która tworzy strumień takich trójkę  $(S_i, T_j, U_k)$ , że  $i \leq j \leq k$ . Użyj procedury `triples` do wygenerowania strumienia wszystkich pitagorejskich trójkę dodatnich liczb całkowitych, tzn. takich trójkę  $(i, j, k)$ , że  $i \leq j$  oraz  $i^2 + j^2 = k^2$ .

### Ćwiczenie 3.70

Dobrze by było móc generować strumienie, w których pary pojawiają się w jakiejś użytecznej kolejności, a nie w kolejności wynikającej z przeplatania wymyślonego ad hoc. Możemy zastosować technikę podobną do procedury `merge` z ćwiczenia 3.56, jeśli zdefiniujemy sposób określania, że jedna para liczb całkowitych jest „mniejsza” od drugiej. Jednym z możliwych rozwiązań jest zdefiniowanie „funkcji wag”  $W(i, j)$  i przyjęcie, że  $(i_1, j_1)$  jest mniejsze niż  $(i_2, j_2)$ , jeśli  $W(i_1, j_1) < W(i_2, j_2)$ . Napisz procedurę `merge-weighted`, która działa tak jak `merge`, ale ma dodatkowy argument `weight`, który oblicza wagi par i jest używany do ustalenia kolejności, w jakiej elementy powinny się pojawiać w wynikowym scalonym strumieniu<sup>69</sup>. Używając jej, napisz uogólnioną wersję procedury `pairs`, o nazwie `weighted-pairs`, której argumentami są dwa strumienie oraz funkcja wagi i która generuje strumień par uporządkowanych ze względu na wagi. Zastosuj tę procedurę do wygenerowania

<sup>69</sup> Wymagamy, aby funkcja wagi była tak określona, że waga pary zwiększa się w miarę, jak przesuwamy się wzduż wiersza lub w dół kolumny w tablicy par.

- (a) strumienia wszystkich par  $(i, j)$  dodatnich liczb całkowitych takich, że  $i \leq j$ , uporządkowanych ze względu na sumę  $i + j$ ;
- (b) strumienia wszystkich par  $(i, j)$  dodatnich liczb całkowitych takich, że  $i \leq j$  oraz ani  $i$ , ani  $j$  nie jest podzielne przez 2, 3 lub 5, uporządkowanych zgodnie z wartościami sum  $2i + 3j + 5ij$ .

### Ćwiczenie 3.71

Liczby, które można przedstawić jako sumy dwóch sześciąników na więcej niż jeden sposób, są czasami nazywane *liczbami Ramanujana*, na cześć matematyka Srinivasa Ramanujana<sup>70</sup>. Uporządkowane strumienie par stanowią eleganckie rozwiązanie problemu obliczania tych liczb. Aby wyznaczyć liczbę, którą można zapisać jako sumę dwóch sześciąników na dwa różne sposoby, wystarczy, że wygenerujemy strumień par liczb całkowitych  $(i, j)$  uporządkowanych zgodnie z wagami  $i^3 + j^3$  (zob. ćwiczenie 3.70), a następnie wyszukamy w strumieniu występujące po sobie pary o tych samych wagach. Napisz procedurę generującą liczby Ramanujana. Pierwszą taką liczbą jest 1729. Jakich jest kolejnych pięć liczb?

### Ćwiczenie 3.72

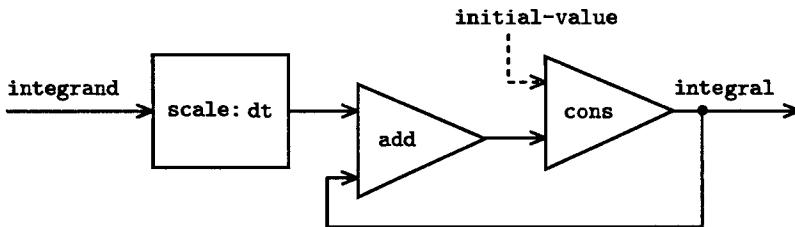
Podobnie jak w ćwiczeniu 3.71, wygeneruj strumień wszystkich liczb, które można przedstawić w postaci sumy dwóch kwadratów na trzy różne sposoby (pokazując sposoby takiego ich przedstawienia).

### Strumienie jako sygnały

Omówienie strumieni rozpoczęliśmy od opisania ich jako obliczeniowego odpowiednika „sygnałów” w systemach przetwarzania sygnałów. W rzeczywistości możemy użyć strumieni do modelowania systemów przetwarzania sygnałów w bardzo bezpośredni sposób — reprezentując wartości sygnału w kolejnych przedziałach czasu jako kolejne elementy strumienia. Możemy na przykład zaimplementować układ *całkujący* lub *sumator*, który dla danego strumienia wejściowego  $x = (x_i)$ , początkowej wartości  $C$  i małego odcinka czasu  $dt$  kumuluje sumę

$$S_i = C + \sum_{j=1}^i x_j dt$$

<sup>70</sup> Zacytujmy wspomnienie G. H. Hardy’ego o Ramanujanie [43]: „Pan Littlewood (jak sądzę) zauważał kiedyś, że «każda dodatnia liczba całkowita była jego przyjacielem». Pamiętam, jak kiedyś wybierałem się, żeby go odwiedzić, gdy leżał chory w Putney. Jechałem taksówką nr 1729 i nadmieniłem, że liczba ta wydała mi się raczej nieciekawa, i miałem nadzieję, że nie jest to zły omen. «Ależ nie», stwierdził, «jest to bardzo ciekawa liczba; jest to najmniejsza liczba, którą można przedstawić jako sumę dwóch sześciąników, na dwa różne sposoby»”. Sztuczkę polegającą na zastosowaniu ważonych par przy generowaniu liczb Ramanujana pokazał nam Charles Leiserson.



Rys. 3.32. Procedura `integral` przedstawiona jako system przetwarzania sygnałów

i generuje strumień wartości  $S = (S_i)$ . Następująca procedura `integral` przy-  
pomina „uwiklaną” definicję strumienia liczb całkowitych (punkt 3.5.2):

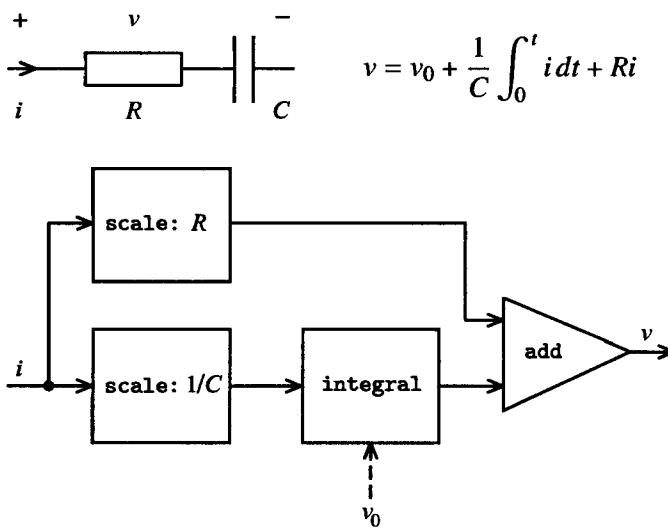
```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
      (add-streams (scale-stream integrand dt)
                    int)))
  int)
```

Na rysunku 3.32 jest przedstawiony schemat systemu przetwarzania sygnałów odpowiadającego procedurze `integral`. Strumień wejściowy jest mnożony przez  $dt$  i przepuszczany przez sumator. Sygnał wyjściowy z sumatora jest przekazywany z powrotem do niego. Występujące w definicji procedury `int` odwoływanie się do samej siebie odzwierciedla przedstawioną na rysunku pętlę sprzężenia zwrotnego łączącą wyjście sumatora z jednym z jego wejść.

### Ćwiczenie 3.73

Możemy modelować obwody elektroniczne, używając strumieni do reprezentowania prądów lub napięć w kolejnych momentach czasu. Przypuśćmy na przykład, że mamy obwód  $RC$  złożony z rezystora o rezystancji  $R$  oraz kondensatora o pojemności  $C$ , połączonych szeregowo. Napięcie  $v$  powstające na wyjściu obwodu w odpowiedzi na prąd  $i$  płynący przez obwód jest określone wzorem podanym na rys. 3.33, którego strukturę przedstawiono na załączonym diagramie przepływu sygnałów.

Napisz procedurę `RC` modelującą ten obwód. Jej argumentami powinny być wartości  $R$ ,  $C$  i  $dt$ . Jej wynikiem zaś powinna być procedura, której argumentami są strumień reprezentujący prąd  $i$  oraz początkowa wartość  $v_0$  napięcia na kondensatorze, a wynikiem jest strumień wartości napięcia  $v$ . Przykładowo, za pomocą procedury `RC` powinieneś móc zamodelować obwód  $RC$  złożony z rezystora o rezystancji  $R = 5 \Omega$  i kondensatora o pojemności  $C = 1 \text{ F}$ , przyjmując półsekundowy kwant czasu, wprowadzając (define `RC1` (`RC` 5 1 0.5)). Tak zdefiniowana procedura `RC1` przekształca strumień reprezentujący ciąg kolejnych wartości prądu i początkowe napięcie na kondensatorze w strumień wartości napięcia na wyjściu.



Rys. 3.33. Obwód RC i odpowiadający mu diagram przepływu sygnałów

### Ćwiczenie 3.74

Liz P. Haker tworzy system przetwarzający sygnały pochodzące z czujników fizycznych. Jedną z istotnych funkcji, jakie chce ona zaimplementować, jest możliwość generowania sygnału opisującego przejścia danego sygnału wejściowego *przez zero*. Oznacza to, że sygnał wynikowy powinien przyjmować wartość: +1 za każdym razem, gdy sygnał wejściowy zmienia wartość z ujemnej na dodatnią; -1 za każdym razem, gdy sygnał wejściowy zmienia wartość z dodatniej na ujemną; 0 we wszystkich pozostałych momentach. (Przymijmy w odniesieniu do sygnału wejściowego, że 0 jest dodatnie). Typowy sygnał wejściowy i odpowiadający mu sygnał przejść przez zero mogą mieć na przykład następującą postać:

... 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 ...  
 ... 0 0 0 0 0 -1 0 0 0 0 1 0 0 ...

W systemie Liz sygnał z czujnika jest reprezentowany w postaci strumienia *sense-data*, a *zero-crossings* to odpowiadający mu strumień przejść przez zero. Liz pisze najpierw procedurę *sign-change-detector*, która porównuje znaki dwóch wartości będących jej argumentami i daje w wyniku, odpowiednio, 0, 1 i -1. Potem Liz tworzy w następujący sposób strumień przejść przez zero:

```
(define (make-zero-crossings input-stream last-value)
  (cons-stream
    (sign-change-detector (stream-car input-stream) last-value)
    (make-zero-crossings (stream-cdr input-stream)
      (stream-car input-stream)))))

(define zero-crossings (make-zero-crossings sense-data 0))
```

Szef Liz, Ewa Lu Ator, zauważa mimo chodem, że program ten jest w przybliżeniu równoważny następującemu programowi, który korzysta z uogólnionej wersji `stream-map` z ćwiczenia 3.50:

```
(define zero-crossings
  (stream-map sign-change-detector sense-data (wyrażenie)))
```

Uzupełnij ten program, podając brakujące `(wyrażenie)`.

### Ćwiczenie 3.75

Niestety, zbudowany przez Liz detektor przejść przez zero z ćwiczenia 3.74 okazał się nieefektywny, gdyż szum zakłócający sygnał z czujnika powodował powstawanie fałszywych przejść przez zero. Jan A. Prawie, specjalista od sprzętu, zaproponował, aby Liz wygładziła sygnał, odfiltrowując zakłócenia przed wyznaczeniem przejść przez zero. Liz posłuchała tej rady i postanowiła wyznaczać przejścia przez zero sygnału powstającego przez uśrednienie każdej wartości pochodzącej z czujnika z wartością ją poprzedzającą. Przedstawia ona problem swojemu asystentowi Ludwikowi Myślielakowi, który próbuje zaimplementować rozwiązanie, modyfikując program Liz w następujący sposób:

```
(define (make-zero-crossings input-stream last-value)
  (let ((avpt (/ (+ (stream-car input-stream) last-value) 2)))
    (cons-stream (sign-change-detector avpt last-value)
      (make-zero-crossings (stream-cdr input-stream)
        avpt))))
```

Nie jest to poprawna implementacja rozwiązania Liz. Znajdź błąd popełniony przez Ludwika i napraw go, nie zmieniając struktury programu. (Wskazówka: Musisz zwiększyć liczbę argumentów `make-zero-crossings`).

### Ćwiczenie 3.76

Ewa Lu Ator skrytykowała rozwiązanie przedstawione przez Ludwika w ćwiczeniu 3.75. Napisany przez niego program nie jest modularny, gdyż jest w nim przemieszane wygładzanie z wykrywaniem przejść przez zero. Na przykład wykrywanie przejść przez zero nie powinno wymagać zmian, gdyby Liz znalazła lepszy sposób polepszania sygnału wejściowego. Pomóż Ludwikowi, pisząc procedurę `smooth`, przekształcającą strumień wejściowy w strumień, którego elementy są średnimi kolejnych dwóch elementów strumienia wejściowego. Następnie użyj `smooth` jako składowej w bardziej modularnej implementacji detektora przejść przez zero.

#### 3.5.4. Strumienie i obliczenia odroczone

Procedura `integral`, przedstawiona na końcu poprzedniego punktu, pokazuje, jak możemy za pomocą strumieni modelować systemy przetwarzania sygnałów zawierające pętle sprzężenia zwrotnego. Pętla sprzężenia zwrotnego obejmująca sumator na rys. 3.32 jest modelowana w ten sposób, że w proce-

durze integral wewnętrzny strumień int jest zdefiniowany za pomocą siebie samego:

```
(define int
  (cons-stream initial-value
    (add-streams (scale-stream integrand dt)
      int)))
```

To, że interpreter potrafi sobie radzić z takimi uwikłanymi definicjami, wynika z tego, że forma specjalna `cons-stream` zawiera w sobie formę `delay`. Bez `delay` interpreter nie mógłby utworzyć strumienia `int`, zanim nie obliczyłby obu argumentów `cons-stream`, co wymagałoby, żeby `int` był już zdefiniowany. Ogólnie mówiąc, rola formy `delay` jest zasadnicza w zastosowaniu strumieni do modelowania systemów przetwarzania sygnałów zawierających pętle. Bez niej musielibyśmy tak formułować modele, że dla każdej składowej przetwarzającej sygnały najpierw byłoby obliczane całe wejście, a dopiero potem mogłoby być generowane wyjście. Wykluczałoby to jakiekolwiek pętle.

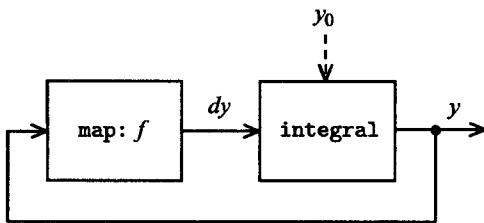
Niestety, strumieniowe modele systemów z pętlami mogą wymagać użycia `delay` nie tylko jako „ukrytego” elementu `cons-stream`. Na rysunku 3.34 jest na przykład przedstawiony system przetwarzania sygnałów służący do rozwiązywania równania różniczkowego  $dy/dt = f(y)$ , gdzie  $f$  jest daną funkcją. Widać tu składową realizującą przekształcenie  $f$  połączoną pętlą sprzężenia zwrotnego z układem całkującym, w sposób bardzo podobny do tego, w jaki łączy się układy komputerów analogowych, których używa się faktycznie do rozwiązywania takich równań.

Zakładając, że znamy początkową wartość  $y_0$  zmiennej  $y$ , moglibyśmy spróbować zamodelować ten system za pomocą następującej procedury:

```
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y))
  y)
```

Procedura ta jednak nie zadziała, gdyż w pierwszym wierszu `solve` odwołanie do `integral` wymaga, aby wejście `dy` było zdefiniowane, co ma miejsce dopiero w drugim wierszu `solve`.

Jednakże myśl przewodnia naszej definicji ma sens, gdyż w zasadzie można zacząć generować strumień `y`, nie znając jeszcze `dy`. Istotnie, `integral` oraz wiele innych operacji na strumieniach mają podobne własności jak `cons-stream` — polegają one na tym, że możemy wygenerować część wyniku na podstawie jedynie częściowych informacji o argumentach. W przypadku `integral` pierwszy element strumienia wyjściowego jest równy parametrowi `initial-value`. Tak więc możemy wygenerować pierwszy element strumienia wyjściowego bez obliczania funkcji podcałkowej `dy`. Gdy już znamy



Rys. 3.34. „Układ komputera analogowego” rozwiązuje równanie  $dy/dt = f(y)$

pierwszy element strumienia  $y$ , wówczas `stream-map` (w drugim wierszu definicji `solve`) może wygenerować pierwszy element  $dy$ , dzięki któremu można wygenerować następny element  $y$  itd.

Wykorzystamy ten pomysł, przedefiniowując `integral` tak, aby całkowany strumień był *argumentem odroczonym*. Procedura `integral` będzie wymuszać (za pomocą `force`) obliczenie wartości funkcji podcałkowej tylko wtedy, gdy będzie to potrzebne do wygenerowania dalszych elementów strumienia wyjściowego, a nie tylko jego pierwszego elementu:

```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
      (let ((integrand (force delayed-integrand)))
        (add-streams (scale-stream integrand dt)
                     int))))
  int)
```

Teraz możemy zaimplementować procedurę `solve`, odraczając w definicji  $y$  obliczenie  $dy$ <sup>71</sup>:

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Mówiąc ogólnie, przy każdym wywołaniu `integral` trzeba użyć `delay` do odroczenia obliczenia całkowanego argumentu. Możemy zademonstrować działanie procedury `solve`, przybliżając liczbę  $e \approx 2,718$  jako rozwiązanie równania różniczkowego  $dy/dt = y$  dla punktu początkowego  $y(0) = 1$ , w punkcie  $y = 1$ :

```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000)
2.716924
```

<sup>71</sup> Procedura ta nie musi działać dla wszystkich implementacji języka Scheme, chociaż w przypadku każdej implementacji możliwa jest jej prosta modyfikacja, która będzie działać. Problem dotyczy subtelnego różnic w sposobach, w jakie implementacje języka Scheme obsługują definicje wewnętrzne. (Zobacz punkt 4.1.6).

### Ćwiczenie 3.77

Użyta powyżej definicja procedury `integral` jest analogiczna do „uwikłanej” definicji nieskończonego strumienia liczb całkowitych z punktu 3.5.2. Można podać alternatywną definicję `integral` przypominającą bardziej definicję `integers-starting-from` (również z punktu 3.5.2):

```
(define (integral integrand initial-value dt)
  (cons-stream initial-value
    (if (stream-null? integrand)
        the-empty-stream
        (integral (stream-cdr integrand)
                  (+ (* dt (stream-car integrand))
                     initial-value)
                  dt)))))
```

Gdy użyjemy tej procedury do modelowania systemów zawierających pętle, wystąpi ten sam problem co przy zastosowaniu pierwotnej wersji `integral`. Zmodyfikuj tę procedurę tak, aby `integrand` był argumentem odroczonym i aby można jej było dzięki temu użyć w powyższej procedurze `solve`.

### Ćwiczenie 3.78

Rozważmy zadanie polegające na skonstruowaniu systemu przetwarzania sygnałów, służącego do badania jednorodnych liniowych równań różniczkowych drugiego rzędu:

$$\frac{d^2y}{dt^2} - a \frac{dy}{dt} - by = 0$$

Strumień wyjściowy modelujący  $y$  jest generowany przez układ zawierający pętlę. Jest tak, ponieważ wartość  $d^2y/dt^2$  zależy od wartości  $y$  i  $dy/dt$ , a obie z nich można określić, całkując  $d^2y/dt^2$ . Diagram, który chcielibyśmy zakodować, jest pokazany na rys. 3.35. Napisz procedurę `solve-2nd`, której argumentami są współczynniki  $a$  i  $b$ , stała  $dt$  oraz początkowe wartości  $y_0$  i  $dy_0$  zmiennych  $y$  i  $dy/dt$  i która generuje strumień kolejnych wartości  $y$ .

### Ćwiczenie 3.79

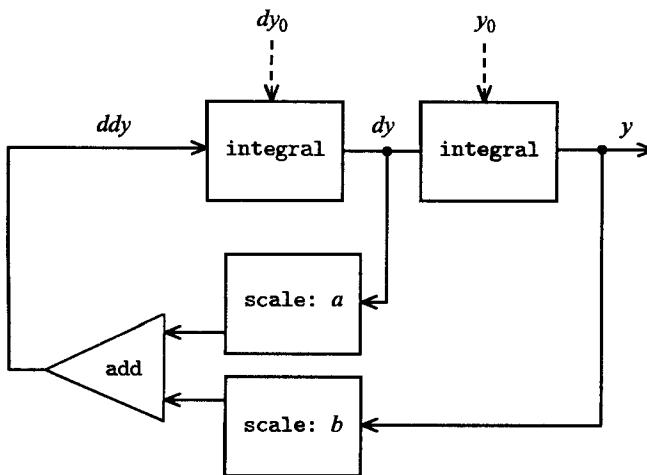
Uogólnij procedurę `solve-2nd` z ćwiczenia 3.78 tak, aby można jej było użyć do rozwiązywania ogólnych równań różniczkowych drugiego rzędu:  $d^2y/dt^2 = f(dy/dt, y)$ .

### Ćwiczenie 3.80

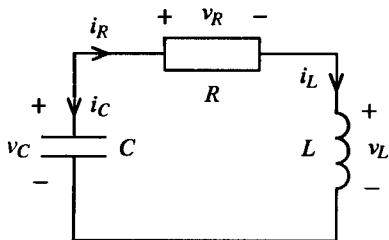
Szeregowy obwód RLC składa się z rezystora, kondensatora i cewki indukcyjnej, połączonych szeregowo, jak to widać na rys. 3.36. Jeśli  $R$ ,  $L$  i  $C$  oznaczają odpowiednio rezystancję, indukcyność i pojemność wymienionych elementów, to zależność między napięciem ( $v$ ) i prądem ( $i$ ) dla tych trzech elementów można opisać następującymi równaniami:

$$v_R = i_R R$$

$$v_L = L \frac{di_L}{dt}$$



Rys. 3.35. Diagram przepływu sygnałów reprezentujący liniowe równanie różniczkowe drugiego rzędu



Rys. 3.36. Szeregowy obwód RLC

$$i_C = C \frac{dv_C}{dt}$$

a połączenia w obwodzie narzucają następujące związki:

$$i_R = i_L = -i_C$$

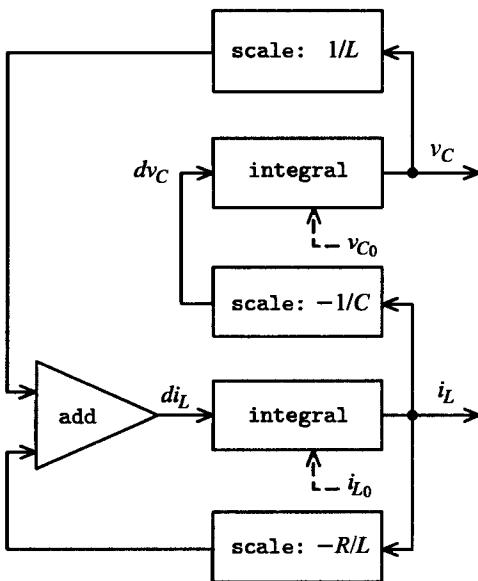
$$v_C = v_L + v_R$$

Łącząc te równania, otrzymujemy, że stan obwodu (określony przez  $v_C$ , czyli napięcie na kondensatorze, oraz  $i_L$ , czyli prąd płynący przez cewkę indukcyjną) jest opisany dwoma równaniami różniczkowymi:

$$\frac{dv_C}{dt} = -\frac{i_L}{C}$$

$$\frac{di_L}{dt} = \frac{1}{L}v_C - \frac{R}{L}i_L$$

Diagram przepływu sygnałów reprezentujący ten układ równań różniczkowych jest pokazany na rys. 3.37.



Rys. 3.37. Diagram przepływu sygnałów reprezentujący układ równań różniczkowych opisujących szeregowy obwód RLC

Napisz procedurę RLC, której argumentami są parametry obwodu  $R$ ,  $L$  i  $C$  oraz kwant czasu  $dt$ . Podobnie jak w przypadku procedury RC z ćwiczenia 3.73, wynikiem procedury RLC powinna być procedura, której argumentami są początkowe wartości  $v_{C_0}$  i  $i_{L_0}$  zmiennych stanu i która daje w wyniku parę (zbudowaną za pomocą `cons`) strumieni wartości  $v_C$  i  $i_L$ . Wygeneruj za pomocą procedury RLC parę strumieni modelujących zachowanie szeregowego obwodu RLC dla  $R = 1 \Omega$ ,  $C = 0,2 \text{ F}$ ,  $L = 1 \text{ H}$ ,  $dt = 0,1 \text{ s}$  oraz początkowych wartości  $i_{L_0} = 0 \text{ A}$  i  $v_{C_0} = 10 \text{ V}$ .

### Normalna kolejność obliczania

Przykłady zamieszczone w niniejszym punkcie pokazują, w jaki sposób jawne użycie `delay` i `force` daje dużą elastyczność programowania, ale te same przykłady pokazują też, jak może to skomplikować programy. Na przykład nowa procedura `integral` umożliwia modelowanie systemów zawierających pętle, musimy jednak teraz pamiętać, że wywołując `integral`, powinniśmy przekazać jej odroczoną funkcję podcałkową, i musimy uwzględnić ten fakt w każdej procedurze używającej `integral`. W rezultacie powstały dwie klasy procedur: zwykłe procedury i procedury o argumentach odroczonych. Ogólnie mówiąc, tworzenie odrębnych klas procedur prowadzi również do powstania odrębnych klas procedur wyższych rzędów<sup>72</sup>.

<sup>72</sup> Jest to tylko drobne odbicie w Lispie trudności, jakie występują w językach ze ścisłą kontrolą typów — takich jak te, które występują w Pascalu z procedurami wyższego rzędu. W językach takich programista musi określić typy danych argumentów oraz wyniku każdej procedury —

Jeden ze sposobów na to, aby uniknąć powstania dwóch różnych klas procedur, polega na tym, żeby wszystkie procedury miały argumenty odroczone. Moglibyśmy przyjąć model obliczeń, w którym obliczanie wartości argumentów wszystkich procedur jest automatycznie odraczane do momentu, gdy ich wartości są faktycznie potrzebne (np. do momentu, gdy wykonujemy na nich operacje pierwotne). Oznaczałoby to zastosowanie w naszym języku normalnej kolejności obliczania, którą opisaliśmy już przy okazji wprowadzania podstawieniowego modelu obliczeń w punkcie 1.1.5. Zmiana kolejności obliczania na normalną pozwala w jednolity i elegancki sposób uprościć zastosowanie obliczeń odroczonych i przyjęcie takiej strategii byłoby naturalne, gdybyśmy zajmowali się jedynie przetwarzaniem strumieni. W podrozdziale 4.2 — po tym, jak zbadamy budowę evaluatora — zobaczymy, jak można w ten sposób zmienić język. Niestety, wprowadzenie odroczeń przy wywoływaniu procedur powoduje zamęt w kolejności zdarzeń, niwcząc tym samym możliwość konstruowania programów, które zależą od kolejności zdarzeń — takich jak programy używające przypisań, modyfikujące dane lub wykonujące operacje wejścia-wyjścia. Nawet pojedyncze `delay` kryjące się w `cons-stream` może wprowadzić spore zamieszanie, co ilustrują ćwiczenia 3.51 i 3.52. Dotychczas nie jest znany dobry sposób połączenia w językach programowania danych modyfikowalnych i obliczeń odroczonych, a poszukiwanie sposobów radzenia sobie z obydwojoma tymi mechanizmami naraz stanowi temat wielu badań.

### 3.5.5. Modularność programów funkcyjnych i modularność obiektów

Jak widzieliśmy w punkcie 3.1.2, jedną z głównych korzyści z wprowadzenia przypisania była możliwość zwiększenia modularności systemów poprzez opakowanie, czyli „ukrycie”, części stanu dużego systemu w zmiennych lokalnych. Modele strumieniowe umożliwiają równoważny stopień modularności bez użycia przypisania. W celu zilustrowania tego możemy powtórnie zaimplementować przybliżenie liczby  $\pi$  metodą Monte Carlo, które badaliśmy w punkcie 3.1.2, rozpatrując je z punktu widzenia przetwarzania strumieni.

Kwestią mającą kluczowy wpływ na modularność programu było ukrycie przed programami używającymi generatora liczb losowych jego wewnętrzne-

---

czy są to liczby, wartości logiczne, ciągi itp. W rezultacie nie możemy wyrazić takich abstrakcji jak „przekształcenie wszystkich elementów ciągu zgodnie z zadaną procedurą `proc`” w postaci pojedynczej procedury wyższego rzędu, takiej jak `stream-map`. Zamiast tego musielibyśmy zdefiniować po jednej procedurze przekształcającej dla każdej kombinacji typów argumentów i wyników `proc`, jakie mogą się pojawić. Zachowanie praktycznego pojęcia „typu danych” w obecności procedur wyższych rzędów powoduje powstanie wielu trudnych problemów. Jeden ze sposobów ich rozwiązania można prześledzić na przykładzie języka ML [34], w którym „polimorficzne typy danych” obejmują szablony typów przekształceń wyższych rzędów. Ponadto typy danych dla większości procedur w języku ML nie są nigdy jawnie deklarowane przez programistę. Zamiast tego ML zawiera mechanizm *wyprowadzania typów*, który na podstawie informacji zawartych w środowisku wywnioskowuje, jaki jest typ nowo definiowanych procedur.

go stanu. Zaczęliśmy od napisania procedury `rand-update`, której kolejne wartości dostarczały nam liczb losowych, i wykorzystaliśmy ją do zbudowania generatora liczb losowych:

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

W sformułowaniu strumieniowym nie ma generatora liczb losowych jako takiego, a jedynie strumień liczb losowych będących wynikami kolejnych wywołań `rand-update`:

```
(define random-numbers
  (cons-stream random-init
    (stream-map rand-update random-numbers)))
```

Używamy go do utworzenia strumienia wyników eksperymentów Cesàro wykonywanych na parach kolejnych elementów strumienia `random-numbers`:

```
(define cesaro-stream
  (map-successive-pairs (lambda (r1 r2) (= (gcd r1 r2) 1))
    random-numbers))

(define (map-successive-pairs f s)
  (cons-stream
    (f (stream-car s) (stream-car (stream-cdr s)))
    (map-successive-pairs f (stream-cdr (stream-cdr s)))))
```

Strumień `cesaro-stream` jest teraz przesyłany do procedury `monte-carlo`, która generuje strumień przybliżonych prawdopodobieństw. Wyniki te są przekształcane na strumień przybliżeń  $\pi$ . Ta wersja programu nie wymaga parametru określającego, ile prób należy wykonać. Lepsze przybliżenia  $\pi$  (polegające na wykonaniu większej liczby eksperymentów) są uzyskiwane przez przeglądanie większego fragmentu strumienia `pi`:

```
(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
      (/ passed (+ passed failed))
      (monte-carlo
        (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
    (next (+ passed 1) failed)
    (next passed (+ failed 1)))))
```

```
(define pi
  (stream-map (lambda (p) (sqrt (/ 6 p)))
              (monte-carlo cesaro-stream 0 0)))
```

Podejście takie jest w znacznym stopniu modularne, gdyż możemy sformułować ogólną procedurę `monte-carlo`, która może operować na dowolnych eksperymentach. A mimo to nie używamy przypisań ani stanów lokalnych.

### Ćwiczenie 3.81

W ćwiczeniu 3.6 omówiliśmy uogólnienie generatora liczb losowych, które umożliwia ustawianie stanu generatora, a przez to generowanie powtarzalnych ciągów liczb „losowych”. Sformułuj strumieniową wersję generatora, który otrzymuje wejściowy strumień zleceń wygenerowania (`generate`) nowej liczby losowej lub ustawienia (`reset`) określonej wartości początkowej ciągu, co powoduje wygenerowanie określonego strumienia liczb losowych. Nie używaj w swoim rozwiążaniu przypisania.

### Ćwiczenie 3.82

Zrób jeszcze raz ćwiczenie 3.5 z całkowania metodą Monte Carlo, tym razem za pomocą strumieni. Strumieniowa wersja procedury `estimate-integral` nie będzie miała argumentu określającego liczbę wykonywanych prób. Zamiast tego będzie ona generować strumień przybliżeń opartych na coraz większej liczbie prób.

## Spojrzenie na czas z punktu widzenia programowania funkcyjnego

Powróćmy do kwestii obiektów i stanów, które poruszyliśmy na początku niestandardowego rozdziału, i przyjrzyjmy się im w nowym świetle. Wprowadziłmy przypisanie i modyfikowalne obiekty danych po to, aby dostarczyć mechanizm umożliwiający modularne budowanie programów modelujących systemy ze stanami. Skonstruowaliśmy obiekty obliczeniowe z lokalnymi zmiennymi stanu i modyfikowalne te zmienne za pomocą przypisania. Modelowaliśmy zachowanie w czasie obiektów ze świata za pomocą zachowania w czasie odpowiadających im obiektów obliczeniowych.

Teraz wiemy już też, że strumienie zapewniają alternatywny sposób modelowania obiektów ze stanami lokalnymi. Możemy modelować zmieniającą się wielkość, taką jak lokalny stan pewnego obiektu, za pomocą strumienia reprezentującego historię kolejnych stanów. W gruncie rzeczy, używając strumieni, reprezentujemy czas wprost, oddzielając bieg czasu w symulowanym świecie od sekwencji zdarzeń zachodzących w trakcie obliczeń. Istotnie, że względem na zastosowanie `delay` związek między symulowanym czasem w modelu a kolejnością zdarzeń zachodzących w trakcie obliczeń może być niewielki.

Aby przeciwstawić sobie te dwa podejścia do modelowania, rozważmy ponownie implementację „procesora wypłat”, który monitoruje saldo konta bankowego. W punkcie 3.1.3 zaimplementowaliśmy uproszczoną wersję takiego procesora:

---

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

Wywołania `make-simplified-withdraw` tworzą obiekty obliczeniowe, z których każdy ma lokalną zmienną stanu `balance`, zmniejszaną przez kolejne wywołania obiektu. Argumentem wywołań obiektu jest kwota (`amount`), a wynikiem nowe saldo konta. Możemy sobie wyobrazić użytkownika systemu bankowego, który wprowadza ciąg danych dla takiego obiektu i obserwuje ciąg wyników wyświetlnych na ekranie.

Alternatywnie, możemy zamodelować procesor wypłat w postaci procedury, której argumentami są saldo i strumień kwot wypłat i która generuje strumień kolejnych sald konta:

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
    balance
    (stream-withdraw (- balance (stream-car amount-stream))
      (stream-cdr amount-stream))))
```

`Stream-withdraw` stanowi implementację dobrze określonej funkcji matematycznej, której wartość jest w pełni określona przez jej argumenty. Przyjmijmy jednak, że strumień wejściowy `amount-stream` jest złożony z kolejnych wartości wprowadzanych przez użytkownika i że strumień kolejnych sald jest wyświetlany na ekranie. Wówczas z punktu widzenia użytkownika, który wprowadza wartości i obserwuje wyniki, proces strumieniowy zachowuje się tak samo jak obiekt utworzony przez `make-simplified-withdraw`. Jednakże w wersji strumieniowej nie występuje przypisanie, nie ma lokalnej zmiennej stanu i w rezultacie nie pojawiają się problemy teoretyczne, które napotkaliśmy w punkcie 3.1.3. A mimo to system ma stan!

To naprawdę niezwykłe. Chociaż `stream-withdraw` implementuje dobrze określzoną funkcję matematyczną, której zachowanie się nie zmienia, użytkownik ma wrażenie, że prowadzi interakcję z systemem, który ma zmieniający się stan. Paradoks ten można rozwiązać, zdając sobie sprawę z tego, że to położenie użytkownika w czasie narzuca systemowi jego stan. Gdyby użytkownik mógł nabierać dystansu do interakcji i pomyśleć w kategoriach strumieni sald, a nie poszczególnych transakcji, system wydałby mu się bezstanowy<sup>73</sup>.

Z punktu widzenia jednej części procesu złożonego inne części zdają się zmieniać w czasie. Mają one ukryty, zmienny w czasie stan lokalny. Jeśli

---

<sup>73</sup> Podobne zjawisko ma miejsce w fizyce — gdy obserwujemy poruszającą się cząstkę, mówimy, że jej położenie (stan) się zmienia. Jednak z punktu widzenia linii świata cząstki żadna zmiana nie zachodzi.

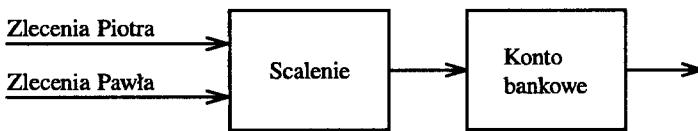
chcemy pisać programy, które modelują za pomocą struktur w komputerze właśnie taki sposób naturalnej dekompozycji naszego świata (tak jak my go postrzegamy jako jego część), to tworzymy obiekty obliczeniowe, które nie są funkcyjne — muszą się zmieniać w czasie. Modelujemy stan za pomocą lokalnych zmiennych stanu, a zmiany stanu modelujemy za pomocą przypisów do tych zmiennych. Postępując w ten sposób, czynimy z czasu wykonania obliczeń model czasu w świecie, którego częścią jesteśmy my sami, i w rezultacie uzyskujemy „obiekty” w naszym komputerze.

Modelowanie przy użyciu obiektów to potężna i w dużej mierze intuicyjna technika, ponieważ odpowiada ona naszemu postrzeganiu interakcji ze światem, którego jesteśmy częścią. Jednakże, jak wielokrotnie widzieliśmy w nimiejszym rozdziale, modele takie są najeżone problemami związanymi z ograniczaniem kolejności zdarzeń i synchronizowaniem wielu procesów. Możliwość uniknięcia tych problemów przyczyniła się do opracowania *funkcyjnych języków programowania* (ang. *functional programming languages*), które nie udostępniają w ogóle przypisania ani danych modyfikowalnych. W takim języku wszystkie procedury implementują dobrze określone funkcje matematyczne swoich argumentów, a ich działanie się nie zmienia. Podejście funkcyjne jest niezwykle atrakcyjne z punktu widzenia systemów współbieżnych<sup>74</sup>.

Jeśli jednak przyjrzymy się dokładnie, to dostrzeżemy, że problemy związane z czasem zakradają się również do modeli funkcyjnych. Jedna ze szczególnie dokuczliwych dziedzin dotyczy konstruowania systemów interakcyjnych, a zwłaszcza takich, które modelują interakcje między niezależnymi jednostkami. Rozważmy na przykład jeszcze raz implementację systemu bankowego umożliwiającego zakładanie wspólnych kont bankowych. W konwencjonalnym systemie z użyciem przypisów i obiektów fakt, że Piotr i Paweł współdzielą konto, modelowalibyśmy, przesyłając ich zlecenia wykonania transakcji do jednego obiektu konta bankowego, jak to widzieliśmy w punkcie 3.1.3. Rozpatrując strumieniowy punkt widzenia, gdy nie ma „obiektów” jako takich, wspominaliśmy już, że można zamodelować konto bankowe jako proces, który operuje na strumieniu zleceń transakcji i generuje strumień wyników. Więc tym razem fakt, że Piotr i Paweł mają wspólne konto bankowe, moglibyśmy zamodelować, scalając strumień zleceń transakcji Piotra i Pawła i przesyłając wynikowy strumień do procesu konta bankowego, jak to widać na rys. 3.38.

Problem z takim sformułowaniem tkwi w pojęciu *scalania*. Nie wystarczy scałić dwa strumienie, biorąc po prostu na przemian po jednym zleceniu Piotra i Pawła. Przypuśćmy, że Paweł korzysta z konta bardzo rzadko. Trudno

<sup>74</sup> John Backus, twórca Fortranu, gdy w 1978 r. odbierał nagrodę ACM im. Turinga, nakreślił przejrzystą wizję programowania funkcyjnego. W swym przemówieniu z okazji otrzymania nagrody [5] silnie popierał podejście funkcyjne. Dobry przegląd programowania funkcyjnego można znaleźć w [17, 47].



Rys. 3.38. Wspólne konto bankowe zamodelowane przez scalenie dwóch strumieni zleceń transakcji

byłoby nam zmusić Piotra, aby czekał, aż Paweł wyda zlecenie, zanim sam będzie mógł wydać kolejne zlecenie. Jakkolwiek takie scalanie jest zaimplementowane, musi ono przeplatać dwa strumienie transakcji w pewien sposób wymuszony przez „czas rzeczywisty” postrzegany przez Piotra i Pawła, w tym sensie, że jeżeli Piotr i Paweł się spotkają, to zgodzą się, że pewne transakcje zostały wykonane przed spotkaniem, a pozostałe transakcje zostały wykonane po spotkaniu<sup>75</sup>. Jest to dokładnie takie samo ograniczenie, z jakim mieliśmy do czynienia w punkcie 3.4.1, gdzie odkryliśmy konieczność wprowadzenia jawnnej synchronizacji w celu zapewnienia „poprawnej” kolejności zdarzeń przy wspólnie przetwarzaniu obiektów ze stanem. Tak więc konieczność scalania danych pochodzących od różnych agentów powstała przy próbie zachowania stylu funkcyjnego, wprowadza te same problemy, które styl funkcyjny miał wyeliminować.

Na początku niniejszego rozdziału postawiliśmy sobie za cel budowanie modeli obliczeniowych mających strukturę odpowiadającą naszemu postrzeganiu świata rzeczywistego, który staramy się modelować. Możemy modelować świat jako zbiór oddzielnych, ograniczonych czasowo, komunikujących się ze sobą obiektów mających stany lub też jako ponadczasową i bezstanową jedność. Każde z tych podejść ma swoje zalety, ale żadne z nich nie jest w pełni zadowalające. Wielka unifikacja jeszcze się nie pojawiła<sup>76</sup>.

<sup>75</sup> Zauważmy, że dla dowolnych dwóch strumieni istnieje na ogół więcej niż jedna dopuszczalna kolejność przeplatania. Tak więc, formalnie rzecz biorąc, „scalanie” jest relacją, a nie funkcją — jej wynik nie jest deterministyczną funkcją danych wejściowych. Wspominaliśmy już (przypis 39), że niedeterminizm jest nieodłączną cechą współbieżności. Relacja scalania przedstawia ten sam zasadniczy niedeterminizm z funkcyjnego punktu widzenia. W podrozdziale 4.3 spojrzymy na niedeterminizm z jeszcze innego punktu widzenia.

<sup>76</sup> Model obiektowy przybliża świat, dzieląc go na oddzielne kawałki. Model funkcyjny nie jest zmodularyzowany zgodnie z granicami obiektów. Model obiektowy jest przydatny wówczas, gdy współdzielone zmienne stanu „obiektów” są znacznie mniejsze od ich niewspółdzielonych stanów. Przykładem dziedziny, w której podejście obiektowe zawodzi, jest mechanika kwantowa, w której myślenie o cząstkach jako o niezależnych bytach prowadzi do paradoksów i pomyłek. Unifikacja podejścia obiektowego i funkcyjnego może w niewielkim stopniu dotyczyć programowania, a raczej dotyczyć podstawowych kwestii epistemologicznych.

# 4

## Abstrakcja metajęzykowa

... Magia tkwi w słowach — Abrakadabra, Sezamie otwórz się i cała reszta — ale zaklęcia z jednej historii nie mają magicznej mocy w innej. Prawdziwa magia polega na zrozumieniu, które słowa działają, kiedy i na co; cała sztuka polega na tym, żeby nauczyć się sztuki.

... A słowa te składają się z liter naszego alfabetu — paru tuzinów zakrętasów, które potrafimy napisać piórem. Oto klucz! A także skarb, gdybyśmy tylko mogli dostać go w swoje ręce! To tak jakby — jakby klucz do skarbu był skarbem!

John Barth, *Chimera*

Badając konstruowanie programów, widzieliśmy, że wytrawni programiści sprawują nadzór nad złożonością swoich konstrukcji, stosując takie same ogólne techniki, jakie stosują konstruktory wszystkich systemów złożonych. Łączą oni elementy pierwotne w celu uzyskania obiektów złożonych, dokonując abstrakcji obiektów złożonych, tworząc elementy składowe wyższych rzędów, oraz zachowując modularność, przyjmując odpowiednią całościową wizję struktury systemu. Przedstawiając te techniki, użyliśmy Lispu jako języka do opisu procesów oraz do tworzenia obliczeniowych obiektów danych i procesów modelujących złożone zjawiska zachodzące w świecie rzeczywistym. Jednakże gdy stawiamy czoło coraz bardziej złożonym problemom, okazuje się, że Lisp czy wręcz dowolny ustalony język programowania przestaje być wystarczający. Chcąc skuteczniej wyrażać pojęcia, musimy nieustannie zwracać się ku nowym językom. Tworzenie nowych języków to potężna metoda panowania nad złożonością tworzonych konstrukcji — często możemy lepiej radzić sobie ze złożonymi problemami, przyjmując nowy język, który pozwala inaczej opisywać problem (a więc i inaczej myśleć o nim), używając pojęć pierwotnych oraz środków łączenia i środków abstrakcji dobranych szczególnie pod kątem danego problemu<sup>1</sup>.

---

<sup>1</sup> Taki sposób postępowania jest rozpowszechniony w naukach inżynierijnych. Na przykład w elektronice używa się wielu różnych języków do opisywania układów. Dwa z nich to język *schematów ideowych* obwodów elektronicznych i język *schematów blokowych* systemów przetwarzania sygnałów. Język schematów ideowych uwypukla fizyczny model urządzeń, opisując

Istnieje mnóstwo języków programowania. Są języki niskiego poziomu, takie jak języki maszynowe poszczególnych rodzajów komputerów. Języki te są wykorzystywane do reprezentacji danych i sterowania na poziomie poszczególnych bitów pamięci i pierwotnych instrukcji (rozkazów) maszynowych. Programista używający języka maszynowego zajmuje się zastosowaniem danego sprzętu do budowy systemów i narzędzi stanowiących efektywne implementacje obliczeń z użyciem ograniczonych zasobów. Języki wysokiego poziomu, wzniesione na fundamentach języków maszynowych, ukrywają zagadnienia związane z reprezentacją danych za pomocą zestawów bitów oraz reprezentacją programów jako ciągów instrukcji pierwotnych. Języki te oferują środki łączenia i abstrakcji, takie jak definicje procedur, odpowiednie do organizacji dużych systemów.

*Abstrakcja metajęzykowa* (ang. *metalinguistic abstraction*) — formułowanie nowych języków — odgrywa ważną rolę we wszystkich gałęziach projektowania technicznego. Jest ona szczególnie ważna w programowaniu komputerów, gdyż w przypadku programowania możemy nie tylko formułować nowe języki, lecz także je implementować, tworząc ewaluatory. *Evaluator* (lub *interpreter*) języka programowania to procedura, która zastosowana do wyrażenia z danego języka wykonuje operacje konieczne do obliczenia wartości tego wyrażenia.

Nie będzie przesadą, jeżeli uznamy następujące stwierdzenie za najbardziej fundamentalną koncepcję programowania:

Evaluator określający znaczenie wyrażeń języka programowania sam jest również programem.

Docenienie tej kwestii oznacza zmianę naszego spojrzenia na nas samych jako programistów. Nagle postrzegamy siebie jako projektantów języków, a nie tylko użytkowników języków zaprojektowanych przez innych.

W rzeczywistości prawie każdy program możemy traktować jako ewaluator pewnego języka. Na przykład system operacji na wielomianach z punktu 2.5.3 zawiera reguły arytmetyki wielomianów i implementuje je za pomocą

---

sposób ich konstrukcji za pomocą poszczególnych elementów. W języku schematów ideowych obiektami pierwotnymi są elementy elektroniczne, takie jak rezistory, kondensatory, cewki indukcyjne czy tranzystory, określone za pomocą wielkości fizycznych zwanych napięciem i prądem. Opisując obwody w języku schematów ideowych, zajmujemy się fizyczną charakterystyką projektu. W przeciwnieństwie do tego obiektami pierwotnymi języka schematów blokowych są moduły przetwarzające sygnały, takie jak filtry i wzmacniacze. Istotna jest przy tym jedynie funkcjonalność modułów, a operując sygnałami, nie martwimy się o ich fizyczną realizację w postaci napięć i prądów. Język schematów blokowych wzniesiono na języku schematów ideowych w tym sensie, że elementy systemów przetwarzania sygnałów są zbudowane z obwodów elektronicznych. Tutaj jednak, rozwiązujeając dany problem konstrukcyjny, zajmujemy się organizacją urządzeń elektronicznych o dużych możliwościach, zakładając przy tym fizyczną wykonalność części składowych. Taki warstwowy zestaw języków stanowi kolejny przykład techniki projektowania wielopoziomowego, którą zilustrowaliśmy na przykładzie języka graficznego przedstawionego w punkcie 2.2.4.

operacji na listowych strukturach danych. Jeśli dołączymy do tego systemu procedury wczytujące i wypisujące wyrażenia wielomianowe, to uzyskamy jądro wyspecjalizowanego języka służącego do symbolicznego rozwiązywania problemów matematycznych. Przedstawiony w punkcie 3.3.4 symulator cyfrowych układów logicznych oraz opisany w punkcie 3.3.5 propagator więzów stanowią pełnoprawne i samodzielne języki, każdy z nich z własnymi pierwotnymi środkami łączenia i środkami abstrakcji. Patrząc z tej perspektywy, technika radzenia sobie z dużymi systemami komputerowymi łączy się z techniką konstruowania nowych języków komputerowych, a sama informatyka staje się niczym więcej (i niczym mniej) niż dyscypliną zajmującą się tworzeniem odpowiednich języków opisowych.

Wybieramy się teraz na przejaźdżkę po technice, w której jedne języki są formułowane za pomocą innych języków. W tym rozdziale będziemy używać Lispu jako podstawy, implementując ewaluatory jako procedury lispowe. Lisp wyjątkowo dobrze nadaje się do tego celu ze względu na możliwości reprezentowania wyrażeń symbolicznych i operowania na nich. Naszym pierwszym krokiem do zrozumienia procesu implementacji języków będzie zbudowanie ewaluatora samego Lispu. Język zaimplementowany przez nasz ewaluator będzie podzioborem języka Scheme — dialekta Lispu, którego używamy w niżej położonej książce. Chociaż ewaluator opisany w tym rozdziale jest napisany dla konkretnego dialekta Lispu, zawiera zasadniczą strukturę ewaluatora dowolnego języka zorientowanego na wyrażenia i przeznaczonego do pisania programów dla maszyny sekwencyjnej. (W rzeczywistości większość procesorów języków zawiera gdzieś głęboko w swoim wnętrzu małeckiego ewaluatora „Lispu”). Na potrzeby niniejszych rozważań oraz ze względów poglądowych ewaluator ten został uproszczony i zostały pominięte pewne funkcje, które powinny być zrealizowane w gotowym systemie lispowym. Pomimo to ten prosty ewaluator nadaje się do wykonywania większości z programów zawartych w tej książce<sup>2</sup>.

Istotną zaletą udostępnienia ewaluatora w postaci programu w Lispie jest to, że można zmieniać reguły ewaluacji, opisując je jako odpowiednie modyfikacje w programie ewaluatora. Możliwości te można z pożytkiem zastosować do uzyskania dodatkowej kontroli nad tym, jak w modelu obliczeniowym jest zrealizowane pojęcie czasu, co było tak istotne w przypadku problemów omawianych w rozdziale 3. Zlagodziliśmy tam trochę stopień złożoności stanów i przypisań, rozdzielając za pomocą strumieni reprezentację biegu czasu w symulowanym systemie od biegu czasu w świecie rzeczywistym. Nasze programy strumieniowe były jednak niekiedy nieporęczne ze względu na ogranicze-

<sup>2</sup> Najważniejsze cechy, jakie pomija ten ewaluator, to mechanizmy sygnalizowania błędów i wspomagania odpluskowania. Bardziej wyczerpujące omówienie ewaluatorów można znaleźć w książce Friedman, Wanda i Haynesa [31], w której przedstawiono języki programowania poprzez szereg ich ewaluatorów napisanych w języku Scheme.

nia wynikające z użytej w języku Scheme stosowanej kolejności obliczania. W podrozdziale 4.2 będziemy wprowadzać zmiany do wykorzystywanego języka w celu uzyskania bardziej eleganckiej jego realizacji, modyfikując evaluator tak, aby udostępniał *normalną kolejność obliczania*.

W podrozdziale 4.3 zaimplementujemy ambitniejszą zmianę językową, dzięki której wyrażenia będą mogły mieć wiele wartości, a nie tylko jedną. W takim języku *obliczeń niedeterministycznych* naturalne jest wyrażanie procesów, które generują wszystkie możliwe wartości wyrażeń, a następnie wyszukiwanie tych wartości, które spełniają określone kryteria. Mówiąc w kategoriach modeli obliczeniowych i czasu, to tak jakbyśmy rozgałęzili czas, tworząc zbiór „możliwych przyszłości”, a następnie szukali odpowiednich linii czasu. Przy zastosowaniu evaluatora niedeterministycznego śledzenie wielu wartości oraz wyszukiwanie odpowiednich z nich są wykonywane automatycznie przez mechanizmy ukryte w języku.

W podrozdziale 4.4 zaimplementujemy język *programowania w logice*, w którym wiedzę wyraża się za pomocą relacji zamiast za pomocą obliczeń o określonych danych wejściowych i wynikach. Chociaż przez to język ten różni się znacznie od Lispu czy nawet od dowolnego konwencjonalnego języka, zobaczymy jednak, że zasadnicza struktura evaluatora języka programowania w logice jest taka sama jak evaluatora Lispu.

## 4.1. Evaluator metacykliczny

Nasz evaluator Lispu będzie zaimplementowany jako program w Lispie. Obliczanie programów w Lispie za pomocą evaluatora, który sam jest zaimplementowany w Lispie, może wydawać się błędym kołem. Jednakże obliczanie jest procesem, a więc opisując proces obliczania, powinniśmy używać Lispu, który jest przecież narzędziem do opisu procesów<sup>3</sup>. Evaluator napisany w tym samym języku, który oblicza, jest nazywany *metacyklicznym* (ang. *metacircular*).

Evaluator metacykliczny jest zasadniczo opisem środowiskowego modelu obliczeń przedstawionego w podrozdziale 3.2, wyrażonym w języku Scheme. Przypomnijmy, że model ten jest określony przez dwie podstawowe reguły:

1. Aby obliczyć wartość kombinacji (wyrażenia złożonego nie będącego formą specjalną), należy obliczyć wartości podwyrażeń, a następnie zastosować wartość podwyrażenia operatora do wartości podwyrażeń argumentów.

<sup>3</sup> Mimo to pewne ważne aspekty procesu obliczania nie będą wyjaśnione przez nasz evaluator. Najważniejsze z nich to szczegółowy mechanizm wywoływanego jednych procedur przez drugie i przekazywanie wyników do procedur wywołujących. Zajmiemy się tymi kwestiami w rozdziale 5, gdzie przyjrzymy się z bliska procesowi obliczania, implementując evaluator jako prostą maszynę rejestrową.

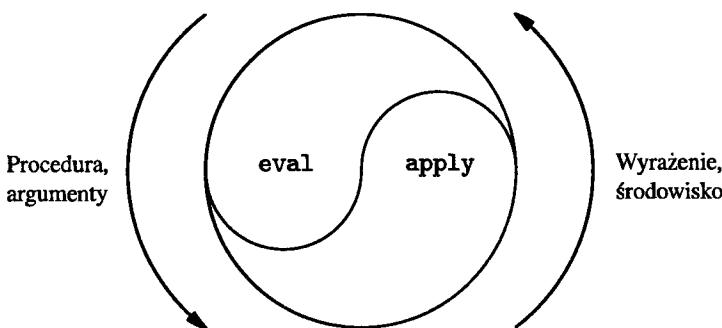
2. Aby zastosować procedurę złożoną do zbioru argumentów, należy obliczyć treść procedury w nowym środowisku. W celu skonstruowania takiego środowiska trzeba rozszerzyć środowiskową część obiektu proceduralnego o ramkę, w której parametry formalne procedury są związane z argumentami, do których procedura jest stosowana.

Te dwie reguły opisują istotę procesu obliczania — podstawowy cykl, w którym wyrażenia, które należy obliczyć w określonych środowiskach, sprowadza się do procedur, które należy zastosować do argumentów, a te z kolei sprawadza się do nowych wyrażeń, które należy obliczyć w nowych środowiskach itd., aż dochodzi się do symboli, których wartości można znaleźć w środowisku, oraz procedur pierwotnych, które można bezpośrednio zastosować (zob. rys. 4.1)<sup>4</sup>. Ten cykl obliczeniowy jest realizowany przez wzajemne oddziaływanie między dwiema kluczowymi procedurami evaluatora: eval i apply, opisanymi w punkcie 4.1.1 (zob. rys. 4.1).

Implementacja evaluatora będzie zależała od procedur definiujących *składnię* obliczanych wyrażeń. Zastosujemy abstrakcję danych, aby uniezależnić evaluator od reprezentacji języka. Zamiast na przykład podejmować decyzję, że przypisanie jest reprezentowane przez listę zaczynającą się od symbolu `set!`, będziemy używać abstrakcyjnego predykatu `assignment?` sprawdzającego, czy mamy do czynienia z przypisaniem, oraz abstrakcyjnych selektorów `assignment-variable` (zmienna przypisania) i `assignment-value` (wartość przypisania) udostępniających odpowiednie części przypisania. Implementacja wyrażeń będzie szczegółowo omówiona w punkcie 4.1.2. Są również operacje, przedstawione w punkcie 4.1.3, określające reprezentację

<sup>4</sup> Jeśli zagwarantujemy możliwość stosowania operacji pierwotnych, to co jeszcze pozostało do zaimplementowania w evaluatorze? Zadaniem evaluatora nie jest określanie znaczenia operacji pierwotnych języka, ale raczej wytworzenie „tkanki łącznej” — środków łączenia i środków abstrakcji — spajającej zestaw operacji pierwotnych w język. Ścisłe mówiąc:

- Evaluator umożliwia posługiwanie się wyrażeniami zagnieżdżonymi. Przykładowo, mimo że proste stosowanie operacji pierwotnych wystarcza do obliczenia wyrażenia `(+ 1 6)`, nie wystarcza ono jednak do obliczenia `(+ 1 (* 2 3))`. Jeśli chodzi o procedurę pierwotną dodawania `+`, to jej argumenty muszą być liczbami; utknęłaby ona, gdybyśmy przekazali jej jako argument wyrażenie `(* 2 3)`. Jedna z ważnych ról evaluatora polega na dyrygowaniu składaniem procedur, tak aby wyrażenie `(* 2 3)` zredukowało się do liczby 6 przed przekazaniem go jako argumentu do `+`.
- Evaluator pozwala używać zmiennych. Na przykład procedura pierwotna dodawania w żaden sposób nie radzi sobie z takimi wyrażeniami jak `(+ x 1)`. Evaluator musi śledzić zmienne i uzyskiwać ich wartości przed wywołaniem procedur pierwotnych.
- Evaluator daje możliwość definiowania procedur złożonych. Wymaga to: pamiętania definicji procedur; określenia, jak używać tych definicji w trakcie obliczania wyrażeń; udostępnienia mechanizmu pozwalającego procedurom na przyjmowanie argumentów.
- Evaluator udostępnia formy specjalne, które muszą być obliczane inaczej niż wywołania procedur.



Rys. 4.1. Cykl eval–apply ukazujący istotę języka programowania

procedur i środowisk. Na przykład `make-procedure` tworzy procedury złożone, `lookup-variable-value` udostępnia wartości zmiennych, a `apply-primitive-procedure` stosuje procedurę pierwotną do zadanej listy argumentów.

#### 4.1.1. Jądro ewaluatora

Proces obliczania może być opisany jako wzajemne oddziaływanie między dwiema procedurami: `eval` (oblicz) i `apply` (zastosuj).

##### Procedura eval

Argumentami procedury `eval` są wyrażenie i środowisko. Klasyfikuje ona wyrażenie i kieruje jego obliczaniem. Struktura `eval` ma postać analizy przypadków — wszystkich rodzajów wyrażeń, jakie mogą pojawić się w obliczeniach. Chcąc, żeby procedura ta była jak najbardziej ogólna, rodzaj wyrażenia ustalamy w abstrakcyjny sposób, nie zakładając żadnej konkretnej reprezentacji poszczególnych rodzajów wyrażeń. Dla każdego rodzaju wyrażeń mamy predykat sprawdzający, czy wyrażenie jest tego rodzaju, oraz abstrakcyjne środki uzyskiwania części wyrażenia. Taka *składnia abstrakcyjna* ułatwia nam zrozumienie, jak możemy zmienić składnię języka, używając tego samego ewaluatora, ale z innym zestawem procedur określających składnię.

##### Wyrażenia pierwotne

- W przypadku *samoobliczających się* wyrażeń, takich jak liczby, wynikiem `eval` jest samo wyrażenie.
- `Eval` musi wyszukiwać wartości zmiennych w środowisku.

##### Formy specjalne

- W przypadku wyrażeń cytowanych wynikiem `eval` jest cytowane wyrażenie.

- Przypisanie do zmiennej (lub jej definicja) wymaga rekurencyjnego wywołania eval w celu obliczenia nowej wartości, która ma być związana ze zmienią. Środowisko musi być tak zmodyfikowane, aby zmienić (lub utworzyć) wiązanie danej zmiennej.
- Wyrażenie `if` wymaga specjalnego przetwarzania jego części, tak aby obliczać następnik, jeśli predykat ma wartość prawda, a alternatywę w przeciwnym razie.
- Lambda-wyrażenie należy przekształcić w dającą się zastosować procedurę, opakowując razem parametry i treść lambda-wyrażenia oraz środowisko obliczenia.
- Wyrażenie `begin` wymaga obliczenia jego wyrażeń składowych w kolejności ich występowania.
- Analiza przypadków (`cond`) jest przekształcana na zagnieżdżone wyrażenia `if`, które następnie są obliczane.

### Kombinacje

- Aby zastosować procedurę, eval musi rekurencyjnie obliczyć operator oraz argumenty kombinacji. Tak otrzymana procedura i argumenty są przekazywane do procedury `apply`, w której dokonuje się faktyczne zastosowanie procedury.

Oto definicja eval:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env))))
        (else
         (error "Nieznany rodzaj wyrażenia -- EVAL" exp))))
```

Dla przejrzystości procedura `eval` została zaimplementowana za pomocą `cond` w formie analizy przypadków. Wadą takiego rozwiązania jest to, że taka procedura obsługuje tylko kilka wyróżnionych rodzajów wyrażeń i nowe rodzaje nie mogą być definiowane bez zmieniania definicji `eval`. W większości implementacji Lispu rozdział ze względu na rodzaj wyrażenia dokonuje się zgodnie ze stylem programowania sterowanego danymi. Pozwala to użytkownikowi na dodawanie nowych rodzajów wyrażeń rozpoznawalnych przez `eval`, bez modyfikowania samej definicji tej procedury. (Zobacz ćwiczenie 4.3).

### Procedura `apply`

Procedura `apply` ma dwa argumenty: procedurę i listę argumentów, do których należy ją zastosować. `Apply` rozróżnia dwa rodzaje procedur. Procedury pierwotne stosuje, wywołując `apply-primitive-procedure`. W przypadku procedur złożonych stosuje je, obliczając kolejne wyrażenia tworzące treść procedury. Środowisko, w którym obliczana jest treść procedury złożonej, jest tworzone jako rozszerzenie środowiska podstawowego, będącego częścią procedury, o ramkę wiążącą parametry procedury z argumentami, do których ma ona być zastosowana. Oto definicja procedury `apply`:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Nieznany rodzaj procedury -- APPLY" procedure))))
```

### Argumenty procedur

Gdy `eval` przetwarza zastosowanie procedury, używa `list-of-values` do uzyskania listy argumentów, do których należy zastosować daną procedurę. Argumentem `list-of-values` są argumenty kombinacji. Procedura ta oblicza każdy z nich i daje w wyniku listę odpowiednich wartości<sup>5</sup>:

<sup>5</sup> Moglibyśmy uprościć klauzulę `application?` w procedurze `eval`, używając `map` (i ustalając, że wynikiem `operands` jest lista), zamiast definiować wprost procedurę `list-of-values`. Zdecydowaliśmy się nie używać tutaj `map`, aby podkreślić fakt, że evaluator może być zaimplementowany bez użycia procedur wyższych rzędów (a co za tym idzie, może być zapisany w języku, który nie zawiera procedur wyższych rzędów), chociaż język, który realizuje, zawiera takie procedury.

---

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

### Wyrażenia warunkowe

Procedura eval-if oblicza w danym środowisku predykatową część wyrażenia if. Jeśli wynikiem jest prawda, to eval-if oblicza następnik; w przeciwnym razie oblicza alternatywę:

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Użycie true? w eval-if zwraca uwagę na powiązanie między językiem implementowanym a językiem implementacji. Wynik if-predicate jest obliczany w języku implementowanym, a zatem wynikiem tego obliczenia jest wartość z tego języka. Predykat interpretera true? tłumaczy tę wartość na wartość, która może być zbadana przez if w języku implementacji — metacykliczna reprezentacja prawdy może nie być taka sama jak kryjąca się za nią reprezentacja prawdy w języku Scheme<sup>6</sup>.

### Ciągi

Procedura eval-sequence jest używana przez apply do obliczania ciągu wyrażeń z treści procedury i przez eval do obliczania ciągu wyrażeń w wyrażeniu begin. Jej argumentami są ciąg wyrażeń i środowisko; oblicza ona dane wyrażenia w kolejności ich występowania. Jej wynikiem jest wartość ostatniego wyrażenia.

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exp exps) env))))
```

### Przypisania i definicje

Następująca procedura realizuje przypisania do zmiennych. Wywołuje ona eval, aby wyznaczyć wartość, która ma być przypisana, po czym przekazuje zmienną i uzyskaną wartość do procedury set-variable-value!, która umieszcza ją w określonym środowisku.

---

<sup>6</sup> W tym przypadku język implementacji i język implementowany są takie same. Kontemplowanie znaczenia predykatu true? poszerza naszą świadomość bez naruszenia jego istoty.

---

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
    (eval (assignment-value exp) env)
    env)
  'ok)
```

Definicje zmiennych są przetwarzane podobnie<sup>7</sup>.

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (eval (definition-value exp) env)
    env)
  'ok)
```

Zdecydowaliśmy tutaj, że wynikiem przypisania lub definicji jest symbol ok<sup>8</sup>.

### Ćwiczenie 4.1

Zwróćmy uwagę, że nie można stwierdzić, czy evaluator metacykliczny oblicza argumenty od lewej do prawej strony, czy też od prawej do lewej. Kolejność obliczania jest dziedziczona po kolejności obliczania leżącej u podstaw implementacji Lispu — jeśli argumenty cons w list-of-values są obliczane od lewej do prawej, to list-of-values oblicza argumenty od lewej do prawej, a jeśli argumenty cons są obliczane od prawej do lewej, to list-of-values oblicza argumenty od prawej do lewej.

Napisz taką wersję list-of-values, która oblicza argumenty od lewej do prawej bez względu na kolejność obliczania w leżącym u podstaw Lispie. Napisz również wersję list-of-values, która oblicza argumenty od prawej do lewej.

#### 4.1.2. Reprezentowanie wyrażeń

Evaluator przypomina program różniczkowania symbolicznego omówiony w punkcie 2.3.2. Oba programy operują na wyrażeniach symbolicznych. W obu programach wynik operacji na wyrażeniu złożonym jest określany przez rekurencyjne operowanie na fragmentach wyrażenia i łączenie wyników w sposób zależny od rodzaju wyrażenia. W obu programach używamy abstrakcji danych, aby oddzielić od siebie ogólne zasady operowania od szczegółów reprezentowania wyrażeń. W przypadku programu różniczkującego oznacza to, że ta sama procedura różniczkowania może operować na wyrażeniach algebraicznych w postaci prefiksowej (notacji polskiej), w postaci infiksowej (tradycyjnej) czy

---

<sup>7</sup> Taka implementacja define pomija subtelną kwestię związaną z obsługą definicji wewnętrznych, chociaż w większości przypadków działa poprawnie. W punkcie 4.1.6 zobaczymy, że problem jednak istnieje, i dowiemy się, jak go można rozwiązać.

<sup>8</sup> Przy okazji wprowadzania define i set! powiedzieliśmy, że wartości będące ich wynikami zależą od implementacji języka Scheme — to znaczy, że implementator może wybrać, jaka wartość ma być wynikiem.

też w jakiejś innej postaci. W przypadku evaluatora oznacza to, że składnia obliczanego języka jest określona wyłącznie przez procedury klasyfikujące i wydzielające fragmenty wyrażeń.

Oto specyfikacja składni naszego języka:

- Jedynymi samoobliczającymi się wyrażeniami są liczby i napisy:

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

- Zmienne są reprezentowane przez symbole:

```
(define (variable? exp) (symbol? exp))
```

- Cytowania mają postać (quote <cytowany tekst>)<sup>9</sup>:

```
(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))
```

Predykat quoted? jest zdefiniowany przy użyciu predykatu tagged-list?, który rozpoznaje listy rozpoczynające się określonym symbolem:

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

- Przypisania mają postać (set! <zmienna> <wartość>):

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

- Definicje mają postać

```
(define <zmienna> <wartość>)
```

---

<sup>9</sup> Jak wspominaliśmy w punkcie 2.3.1, evaluator widzi cytowane wyrażenie jako listę rozpoczętą się od quote, nawet jeśli wyrażenie jest zapisane z apostrofem. Na przykład wyrażenie 'a byłoby postrzegane przez evaluator jako (quote a). Zobacz ćwiczenie 2.55.

lub

```
(define ⟨zmienna⟩ ⟨parametr1⟩ ... ⟨parametrn⟩)
  ⟨treść⟩)
```

Ta druga postać (standardowa definicja procedur) jest lukrem syntaktycznym równoważnym

```
(define ⟨zmienna⟩
  (lambda ⟨⟨parametr1⟩ ... ⟨parametrn⟩⟩)
  ⟨treść⟩))
```

Następujące procedury określają składnię:

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
    (cadr exp)
    (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
    (caddr exp)
    (make-lambda (cdadr exp) ; parametry formalne
                 (cddr exp)))) ; treść
```

- Lambda-wyrażenia to listy rozpoczynające się symbolem `lambda`:

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))
```

Udostępniamy również konstruktor lambda-wyrażeń, używany powyżej przez procedurę `definition-value`:

```
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

- Wyrażenia warunkowe zaczynają się od `if` i zawierają predykat, następnik oraz (opcjonalnie) alternatywę. Jeśli w wyrażeniu nie ma alternatywy, to traktujemy je tak, jakby alternatywą było `false`<sup>10</sup>.

<sup>10</sup> Gdy predykat jest fałszywy i nie ma alternatywy, wartość wyrażenia `if` w języku Scheme jest nieokreślona. Zdecydowaliśmy tutaj, że będzie ona fałszem. Pozwolimy na używanie zmiennych `true` i `false` w obliczanych wyrażeniach, wiążąc je w środowisku globalnym. Zobacz punkt 4.1.4.

```
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

Udostępniamy również konstruktor wyrażeń `if`, którego będziemy używać w `cond->if` do przekształcania wyrażeń `cond` w wyrażenia `if`:

```
(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

- `Begin` ujmuje ciąg wyrażeń w jedno wyrażenie. Udostępniamy operacje składniowe na wyrażeniach `begin`, określające faktyczny ciąg tworzący takie wyrażenia, oraz selektory przekazujące pierwsze wyrażenie i pozostałe wyrażenia w ciągu<sup>11</sup>:

```
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))
```

Definiujemy również konstruktor `sequence->exp` (do wykorzystania w `cond->if`), który przekształca ciąg wyrażeń w jedno wyrażenie, używając `begin`, jeśli to konieczne:

```
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq)))))

(define (make-begin seq) (cons 'begin seq))
```

---

<sup>11</sup> Te selektory operujące na listach wyrażeń — i odpowiadające im selektory operujące na listach argumentów — nie mają stanowić abstrakcji danych. Wprowadzono je jako łatwe do zapamiętania nazwy podstawowych operacji na listach, w celu ułatwienia zrozumienia evaluatora z jawnie określonym sterowaniem, przedstawionego w podrozdziale 5.4.

- Zastosowanie procedury `to` dowolne wyrażenie złożone, innego rodzaju niż wyrażenia opisane powyżej. `Car` wyrażenia to operator, a `cdr` to lista argumentów:

```
(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

### Wyrażenia pochodne

Pewne formy specjalne w naszym języku mogą być zdefiniowane w postaci wyrażeń za pomocą innych form specjalnych, a nie wprost. Jednym z przykładów tego jest `cond`, które może być zaimplementowane jako zagnieżdżone wyrażenia `if`. Możemy na przykład sprowadzić problem obliczania wyrażenia

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

do problemu obliczania następującego wyrażenia zawierającego wyrażenia `if` i `begin`:

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero)
               0)
        (- x)))
```

Taka implementacja obliczania `cond` upraszcza konstrukcję evaluatora, gdyż zmniejsza liczbę form specjalnych, dla których proces obliczania musi być określony wprost.

Udostępniamy procedury składniowe wydzielające części wyrażenia `cond` oraz procedurę `cond->if`, która przekształca wyrażenia `cond` w wyrażenia `if`. Analiza przypadków zaczyna się od `cond`, po czym następuje lista klauzul postaci predykat-akcja. Klauzula jest klauzulą `else`, jeśli jej predykat jest symbolem `else`<sup>12</sup>.

---

<sup>12</sup> Gdy żaden z predykatów nie jest spełniony i nie ma klauzuli `else`, wartość wyrażenia `cond` w języku Scheme nie jest określona — przyjęliśmy tutaj, że będzie to fałsz.

```

(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false ; brak klauzuli else
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error
                  "Klauzula ELSE nie jest ostatnia -- COND->IF"
                  clauses))
            (make-if (cond-predicate first)
                  (sequence->exp (cond-actions first))
                  (expand-clauses rest)))))))

```

Wyrażenia (takie jak `cond`), które implementujemy w postaci transformacji składniowych, nazywamy *wyrażeniami pochodnymi* (ang. *derived expressions*). Wyrażenia `let` są również wyrażeniami pochodnymi (zob. ćwiczenie 4.6)<sup>13</sup>.

### Ćwiczenie 4.2

Ludwik Myśliścielak chce zmienić kolejność klauzul `cond` w procedurze `eval` tak, aby klauzula opisująca stosowanie procedur pojawiła się przed klauzulą opisującą przypisania. Argumentuje on, że dzięki temu interpreter będzie bardziej efektywny — ponieważ programy zawierają zwykle więcej wywołań procedur niż przypisań, definicji itp., więc zmodyfikowana wersja `eval` sprawdzi zwykle mniej klauzul niż oryginalna procedura `eval`, zanim rozpozna rodzaj wyrażenia.

<sup>13</sup> Rzeczywiste systemy lispowe udostępniają mechanizm umożliwiający dodawanie nowych wyrażeń pochodnych i określanie ich implementacji jako transformacji składniowych, bez modyfikowania evaluatora. Taka transformacja zdefiniowana przez użytkownika jest nazywana *makrodefinicją*. Mimo że łatwo można dodać podstawowy mechanizm realizujący makrodefinicje, w powstałyym języku występują subtelne problemy związane z konfliktami nazw. Wiele badań poświęcono poszukiwaniu mechanizmu makrodefinicji, który nie powodowałby takich trudności. Zobacz na przykład [14, 42, 60].

- (a) Gdzie tkwi błąd w pomyśle Ludwika? (Wskazówka: Co zrobi evaluator Ludwika z wyrażeniem `(define x 3)`?)
- (b) Ludwik zdenerwował się tym, że jego pomysł nie wypalił. Jest gotów na wszystko, byleby jego evaluator rozpoznawał wywołania procedur przed sprawdzaniem większości wszystkich pozostałych rodzajów wyrażeń. Pomóż mu, tak zmieniając składnię obliczanego języka, aby wywołania procedur zaczynały się od słowa `call`. Na przykład zamiast `(factorial 3)` musielibyśmy napisać `(call factorial 3)`, a zamiast `(+ 1 2)` musielibyśmy napisać `(call + 1 2)`.

### Ćwiczenie 4.3

Napisz ponownie procedurę `eval`, tak aby rozdział dokonywał się w stylu programowania sterowanego danymi. Porównaj wynik z procedurą różniczkowania sterowaną danymi z ćwiczenia 2.73. (W przypadku składni zaimplementowanej w niniejszym podrozdziale możesz użyć `car` wyrażenia złożonego jako rodzaju wyrażenia).

### Ćwiczenie 4.4

Przypomnijmy sobie definicje form specjalnych `and` i `or` z rozdziału 1:

- `and`: Podwyrażenia są obliczane kolejno od lewej do prawej. Jeśli którykolwiek z nich okaże się fałszywe, to kolejne podwyrażenia nie są obliczane, a wartością całego wyrażenia jest fałsz. Jeśli wszystkie podwyrażenia są prawdziwe, to wartością całego wyrażenia `and` jest wartość ostatniego z nich. Jeśli brak podwyrażeń, to wartością wyrażenia jest prawda.
- `or`: Podwyrażenia są obliczane kolejno od lewej do prawej. Jeśli którykolwiek z nich okaże się prawdziwe, to jego wartość jest wartością całego wyrażenia `or`, a kolejne podwyrażenia nie są obliczane. Jeśli wszystkie podwyrażenia są fałszywe lub brak jest podwyrażeń, to wartością całego wyrażenia `or` jest fałsz.

Zainstaluj w evaluatorze `and` i `or` jako nowe formy specjalne, definiując odpowiednie procedury składniowe i procedury obliczające `eval-and` (`oblicz and`) i `eval-or` (`oblicz or`). Pokaż, jak można alternatywnie zaimplementować `and` i `or` jako wyrażenia pochodne.

### Ćwiczenie 4.5

Scheme dopuszcza dodatkową składnię klauzul `cond`:  $\langle\langle\text{warunek}\rangle\rangle \Rightarrow \langle\text{odbiorca}\rangle$ . Jeśli w wyniku obliczenia  $\langle\text{warunku}\rangle$  uzyskujemy prawdę, czyli wartość różną od fałszu, to obliczany jest  $\langle\text{odbiorca}\rangle$ . Jego wartością musi być procedura jednoargumentowa; jest ona wówczas wywoływana, jej argumentem jest wartość  $\langle\text{warunku}\rangle$ , a jej wynik jest wartością całego wyrażenia `cond`. Na przykład wynikiem

```
(cond ((assoc 'b '((a 1) (b 2))) => cdr)
      (else false))
```

jest liczba 2. Zmodyfikuj obsługę wyrażeń `cond` tak, aby uwzględniała to rozszerzenie składni.

**Ćwiczenie 4.6**

Wyrażenia `let` są wyrażeniami pochodnymi, gdyż

```
(let ((<v1> <e1>) ... (<vn> <en>))
  <treść>)
```

jest równoważne

```
((lambda (<v1> ... <vn>)
  <treść>)
 <e1>
 :
 <en>)
```

Zaimplementuj transformację składniową `let->combination` sprowadzającą obliczenie wyrażenia `let` do obliczenia kombinacji tego rodzaju co pokazana powyżej i dodaj do procedury `eval` odpowiednią klauzulę obsługującą wyrażenia `let`.

**Ćwiczenie 4.7**

Forma `let*` jest podobna do `let`, przy czym zmienne wyrażenia `let*` są wiązane kolejno od lewej do prawej, a każde wiązanie jest tworzone w środowisku, w którym wszystkie poprzedzające wiązania są widoczne. Na przykład wynikiem

```
(let* ((x 3)
      (y (+ x 2))
      (z (+ x y 5)))
  (* x z))
```

jest liczba 39. Wyjaśnij, jak można zapisywać wyrażenia `let*` w postaci wielu zagnieżdżonych wyrażeń `let`, i napisz procedurę `let*->nested-lets` wykonującą takie przekształcenie. Gdybyśmy mieli już zaimplementowaną formę `let` (ćwiczenie 4.6) i chcielibyśmy rozszerzyć evaluator o formę `let*`, to czy wystarczyłoby dodać do procedury `eval` klauzulę, której akcja jest postaci

```
(eval (let*->nested-lets exp) env)
```

czy też musielibyśmy wprost rozwinąć `let*`, używając wyrażeń nie będących wyrażeniami pochodnymi?

**Ćwiczenie 4.8**

„Nazwany `let`” jest wariantem formy `let` postaci

```
(let <zmienna> <wiązania> <treść>)
```

`<Wiązania>` i `<treść>` są takie jak w zwykłym `let`, ale `<zmienna>` jest związana w `<treści>` z procedurą, której treścią jest `<treść>` i której parametrami są zmienne określone przez `<wiązania>`. Tak więc można wielokrotnie wykonywać `<treść>`, wywołując procedurę, której nazwą jest `<zmienna>`. Na przykład procedura iteracyjna obliczająca

liczby Fibonacciego (punkt 1.2.2) może być zapisana przy użyciu nazwanego `let` w następujący sposób:

```
(define (fib n)
  (let fib-iter ((a 1)
                (b 0)
                (count n))
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1)))))
```

Zmodyfikuj procedurę `let->combination` z ćwiczenia 4.6 tak, aby uwzględniała nazwane `let`.

### Ćwiczenie 4.9

Wiele języków udostępnia rozmaite konstrukcje iteracyjne (pętle takie, jak `do`, `for`, `while` i `until`). W języku Scheme procesy iteracyjne można wyrażać za pomocą zwykłych wywołań procedur, a zatem specjalne konstrukcje iteracyjne nie zwiększą zasadniczo mocy obliczeniowej. Jednakże konstrukcje takie są często wygodne. Zaprojektuj kilka konstrukcji iteracyjnych, podaj przykłady ich użycia i pokaż, jak je zaimplementować jako wyrażenia pochodne.

### Ćwiczenie 4.10

Stosując abstrakcję danych, możemy napisać procedurę `eval`, która jest niezależna od konkretnej składni obliczanego języka. Aby to zilustrować, zaprojektuj i zrealizuj nową składnię języka Scheme, modyfikując procedury przedstawione w niniejszym punkcie, ale nie zmieniając `eval` ani `apply`.

#### 4.1.3. Struktury danych evaluatora

Oprócz definicji zewnętrznej składni wyrażeń implementacja evaluatora musi również definiować wewnętrzną strukturę danych, na której evaluator operuje w trakcie wykonywania programów, taką jak reprezentacja procedur i środowisk oraz reprezentacja prawdy i fałszu.

#### Badanie predykatów

Jeśli chodzi o warunki logiczne, to przyjmujemy, że wszystko, co nie jest obiektem `false`, jest prawdą.

```
(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))
```

## Reprezentowanie procedur

Aby móc wykonywać operacje pierwotne, zakładamy, że są dostępne następujące procedury:

- `(apply-primitive-procedure <proc> <argumenty>)`  
stosuje daną procedurę pierwotną `<proc>` do wartości argumentów z listy `<argumenty>`; jej wynikiem jest wynik wywołania danej procedury pierwotnej.
- `(primitive-procedure? <proc>)`  
sprawdza, czy `<proc>` jest procedurą pierwotną.

Te mechanizmy obsługi operacji pierwotnych są dalej opisane w punkcie 4.1.4.

Procedury złożone konstruujemy na podstawie parametrów, treści procedury i środowiska za pomocą konstruktora `make-procedure`:

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))
```

## Operacje na środowiskach

Implementacja evaluatora wymaga implementacji operacji na środowiskach. Jak wyjaśniliśmy to w podrozdziale 3.2, środowisko to ciąg ramek, a każda ramka to tablica wiązań przyporządkowujących zmiennym odpowiadające im wartości. Używamy następujących operacji na środowiskach:

- `(lookup-variable-value <zmienna> <środowisko>)`  
daje w wyniku wartość związaną z symbolem `<zmienna>` w środowisku `<środowisko>` lub zgłasza błąd, jeśli zmienna jest wolna.
- `(extend-environment <zmienne> <wartości> <środowisko otaczające>)`  
daje w wyniku nowe środowisko składające się z nowej ramki, w której symbole z listy `<zmienne>` są związane z odpowiednimi elementami listy `<wartości>`, oraz `<środowiska otaczającego>`.
- `(define-variable! <zmienna> <wartość> <środowisko>)`  
dodaje do pierwszej ramki w środowisku `<środowisko>` nowe wiązanie, które przyporządkowuje zmiennej `<zmienna>` wartość `<wartość>`.

- **(set-variable-value! <zmienna> <wartość> <środowisko>)**

zmienia wiązanie zmiennej **<zmienna>** w środowisku **<środowisko>** tak, że zmienna jest teraz związana z wartością **<wartość>**, lub zgłasza błąd, jeśli dana zmienna jest wolna.

Aby zaimplementować te operacje, będziemy reprezentować środowisko jako listę ramek. Środowiskiem otaczającym środowiska jest **cdr** listy. Środowisko puste to po prostu lista pusta.

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())
```

Każda ramka środowiska jest reprezentowana jako para list: lista zmiennych związanych przez tę ramkę oraz lista odpowiadających im wartości<sup>14</sup>.

```
(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

Aby rozszerzyć środowisko o nową ramkę wiążącą zmienne z wartościami, tworzymy ramkę składającą się z listy zmiennych i listy wartości oraz dołączamy ją do danego środowiska. Jeśli liczba zmiennych nie odpowiada liczbie wartości, to zgłaszany jest błąd.

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Podano zbyt dużo wartości" vars vals)
          (error "Podano zbyt mało wartości" vars vals))))
```

Aby sprawdzić wartość zmiennej w środowisku, przeszukujemy listę zmiennych w pierwszej ramce. Jeśli znajdziemy poszukiwaną zmienną, przekazujemy odpowiadający jej element z listy wartości. Jeśli nie znajdziemy zmiennej

<sup>14</sup> W poniższym kodzie ramki nie stanowią faktycznie abstrakcji danych — procedury **set-variable-value!** i **define-variable!** używają **set-car!**, bezpośrednio modyfikując wartości przechowywane w ramce. Procedury operujące na ramkach zdefiniowano po to, aby łatwiej się czytało kod procedur operujących na środowiskach.

w bieżącej ramce, przeszukujemy środowisko otaczające itd. Jeśli dojdziemy do pustego środowiska, zgłaszamy błąd „zmienna wolna”.

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
              (car vals))
            (else (scan (cdr vars) (cdr vals)))))
      (if (eq? env the-empty-environment)
          (error "Zmienna wolna" var)
          (let ((frame (first-frame env)))
            (scan (frame-variables frame)
                  (frame-values frame))))
      (env-loop env)))
```

Aby ustawić nową wartość zmiennej w określonym środowisku, szukamy danej zmiennej, tak jak w procedurze `lookup-variable-value`, i gdy ją znajdziemy, zmieniamy odpowiadającą jej wartość.

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
              (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
      (if (eq? env the-empty-environment)
          (error "Zmienna wolna -- SET!" var)
          (let ((frame (first-frame env)))
            (scan (frame-variables frame)
                  (frame-values frame))))
      (env-loop env)))
```

Aby zdefiniować zmienną, sprawdzamy, czy w pierwszej ramce istnieje wiązanie dla tej zmiennej, i jeśli tak jest, zmieniamy to wiązanie (tak jak w procedurze `set-variable-value!`). Jeśli takie wiązanie nie istnieje, to wstawiamy je do pierwszej ramki.

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              (add-binding-to-frame! var val frame))))
```

```

((eq? var (car vars))
  (set-car! vals val))
  (else (scan (cdr vars) (cdr vals))))))
  (scan (frame-variables frame)
    (frame-values frame))))

```

Opisana tutaj metoda stanowi tylko jeden z wielu możliwych do przyjęcia sposobów reprezentowania środowisk. Ponieważ zastosowaliśmy zasadę abstrakcji danych, oddzielając konkretnie wybraną reprezentację od reszty evaluatora, więc gdybyśmy chcieli, moglibyśmy zmienić reprezentację środowisk. (Zobacz ćwiczenie 4.11). W rzeczywistych systemach lispowych szybkość wykonywanych przez evaluator operacji na środowiskach — zwłaszcza wyszukiwanie wartości zmiennych — ma decydujący wpływ na wydajność systemu. Opisana tutaj reprezentacja, choć koncepcyjnie prosta, jest nieefektywna i zwykle nie znalazłaby zastosowania w rzeczywistych systemach<sup>15</sup>.

### Ćwiczenie 4.11

Zamiast reprezentować ramki jako pary list, możemy reprezentować je jako listy wiązań, gdzie każde wiązanie jest parą nazwa-wartość. Przepisz operacje na środowiskach tak, aby używały tej alternatywnej reprezentacji.

### Ćwiczenie 4.12

Procedury `set-variable-value!`, `define-variable!` i `lookup-variable-value` można wyrazić za pomocą bardziej abstrakcyjnych procedur przechodzących strukturę środowiska. Zdefiniuj abstrakcje ujmujące wspólne schematy i zdefiniuj za nich pomocą jeszcze raz te trzy procedury.

### Ćwiczenie 4.13

Scheme umożliwia tworzenie nowych wiązań zmiennych za pomocą `define`, ale nie umożliwia usuwania ich. Zaimplementuj w evaluatorze formę specjalną `make-unbound!`, która usuwa wiązanie danego symbolu ze środowiska, w którym wyrażenie `make-unbound!` jest obliczane. Ten problem nie jest sprecyzowany do końca. Na przykład, czy należy usuwać wiązanie tylko z pierwszej ramki środowiska? Uzupełnij specyfikację problemu, uzasadniając wszystkie dokonane wybory.

#### 4.1.4. Uruchamianie evaluatora jako programu

Mając evaluator, mamy w rękach opis (wyrażony w Lispie) procesu, za pomocą którego obliczane są wyrażenia lispowe. Jedną z korzyści z wyrażania evaluatora jako programu jest to, że możemy go uruchomić. Otrzymujemy

<sup>15</sup> Wada tej reprezentacji (jak i jej wariantu z ćwiczenia 4.11) polega na tym, że evaluator może być zmuszony do przeszukiwania wielu ramek w celu znalezienia wiązania danej zmiennej. (Takie podejście jest nazywane *głębokim wiązaniem*). Jeden ze sposobów uniknięcia takiej nieefektywności polega na zastosowaniu strategii zwanej *adresowaniem leksykalnym*, którą omówimy w punkcie 5.5.6.

w ten sposób działający w Lispie model tego, jak Lisp sam oblicza wyrażenia. Może nam to posłużyć jako poligon do eksperymentowania z regułami obliczania, czym zajmiemy się dalej w niniejszym rozdziale.

Nasz program evaluatora sprowadza ostatecznie wyrażenia do zastosowań procedur pierwotnych. Zatem wszystko, czego nam brakuje do uruchomienia evaluatora, to mechanizm, który zleca działającemu pod spodem systemowi lispowemu wykonanie procedur pierwotnych.

Dla każdej nazwy procedury pierwotnej musi istnieć wiązanie, ażeby procedura `eval`, obliczając operator zastosowania operacji pierwotnej, znalazła obiekt, który należy przekazać do procedury `apply`. Tak więc tworzymy środowisko globalne, które nazwom procedur pierwotnych, mogącym pojawić się w obliczanych wyrażeniach, przyporządkowuje niepowtarzalne obiekty. Środowisko globalne zawiera również wiązania symboli `true` i `false`, a zatem można ich używać jako zmiennych w obliczanych wyrażeniach.

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))

(define the-global-environment (setup-environment))
```

Nie ma znaczenia, jak będziemy reprezentować obiekty procedur pierwotnych, dopóki procedura `apply` będzie potrafiła je odróżnić i stosować, używając procedur `primitive-procedure?` i `apply-primitive-procedure`. Zdecydowaliśmy się reprezentować procedury pierwotne jako listy zaczynające się symbolem `primitive` i zawierające procedury z działającego pod spodem Lispu, które implementują dane operacje pierwotne.

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))
```

Procedura `setup-environment` pobiera nazwy operacji pierwotnych i implementujące je procedury z listy<sup>16</sup>:

<sup>16</sup> Na potrzeby evaluatora metacyklicznego dowolna procedura z działającego pod spodem Lispu może być użyta jako pierwotna. Nazwa zainstalowanej operacji pierwotnej nie musi być taka sama jak nazwa jej implementacji. W tym przypadku nazwy są takie same, gdyż evaluator metacykliczny implementuje język Scheme za pomocą niego samego. Tak więc na liście procedur pierwotnych moglibyśmy wymienić np. (`list 'first car`) lub (`list 'square (lambda (x) (* x x))`).

```

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?))
  (dalsze operacje pierwotne)
 ))

(define (primitive-procedure-names)
  (map car
    primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
    primitive-procedures))

```

Aby zastosować procedurę pierwotną do argumentów, stosujemy po prostu jej implementację, używając działającego pod spodem systemu lispowego<sup>17</sup>:

```

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
    (primitive-implementation proc) args))

```

Dla wygody uruchamiania evaluatora metacyklicznego udostępniamy *pętlę sterującą* (ang. *driver loop*), która modeluje pętlę wczytaj-oblicz-wypisz w działającym pod spodem systemie lispowym. Wypisuje ona *znak zachęty* (ang. *prompt*), wczytuje wyrażenie wejściowe, oblicza to wyrażenie w środowisku globalnym i wypisuje wyniki. Każdy wypisywany wynik poprzedzamy *znakiem systemu* (ang. *output prompt*), aby odróżnić wartości wyrażeń od innych danych, które mogą być wypisywane<sup>18</sup>.

<sup>17</sup> Procedura *apply-in-underlying-scheme* to procedura pierwotna *apply* z poprzednich rozdziałów. Procedura *apply* evaluatora metacyklicznego (punkt 4.1.1) modeluje jej działanie. Posiadanie dwóch różnych rzeczy o nazwie *apply* prowadzi do problemów w uruchomieniu evaluatora metacyklicznego, ponieważ definicja procedury *apply* evaluatora zasłania definicję operacji pierwotnej. Jeden ze sposobów obejścia tego problemu polega na przemianowaniu metacyklicznej procedury *apply* w celu uniknięcia konfliktu nazw z operacją pierwotną. Zamiast tego założyliśmy, że odniesienie do pierwotnego *apply* zachowano za pomocą definicji

```
(define apply-in-underlying-scheme apply)
```

występującej przed definicją metacyklicznego *apply*. Pozwala to na odwoływanie się do oryginalnej wersji *apply* pod inną nazwą.

<sup>18</sup> Procedura pierwotna *read* czeka na dane wejściowe od użytkownika i przekazuje kolejne pełne wyrażenie, które wprowadzono. Jeżeli użytkownik napisze na przykład (+ 23 x), to wynikiem *read* jest trójelementowa lista zawierająca symbol +, liczbę 23 i symbol x. Jeżeli użytkownik wprowadzi 'x, to wynikiem *read* jest dwuelementowa lista złożona z symboli *quote* i x.

---

```

(define input-prompt ";;; M-Eval wprowadź wyrażenie:")
(define output-prompt ";;; M-Eval wartość:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

```

Używamy specjalnej procedury wypisującej `user-print`, aby uniknąć wypisywania części środowiskowej procedury złożonej, która to część może być bardzo długą listą (lub może nawet zawierać cykle).

```

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))

```

Teraz do uruchomienia evaluatora potrzebujemy jedynie zainicjować środowisko globalne i rozpoczęć pętlę sterującą. Oto przykładowa interakcja:

```

(define the-global-environment (setup-environment))

(driver-loop)

;;; M-Eval wprowadź wyrażenie:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; M-Eval wartość:
ok ·

;;; M-Eval wprowadź wyrażenie:
(append '(a b c) '(d e f))
;;; M-Eval wartość:
(a b c d e f)

```

## Ćwiczenie 4.14

Ewa Lu Ator i Ludwik Myślicielak eksperymentują z ewaluatorem metacyklicznym. Ewa wprowadziła definicję `map` i uruchomiła kilka testowych programów używających tej procedury. Działyły świetnie. Ludwik z kolei zainstalował wersję systemu, w którym `map` jest operacją pierwotną ewaluatora metacyklicznego. Gdy wypróbował go, wszystko poszło fatalnie. Wyjaśnij, dlaczego `map` Ludwika nie działa, mimo że `map` Ewy działa.

### 4.1.5. Dane jako programy

W myśleniu o programie w Lispie obliczającym wyrażenia lisowe może nam pomóc analogia. Patrząc na program od strony wykonywanych przez niego operacji, możemy uznać, że program to opis (być może nieskończanie dużej) maszyny abstrakcyjnej. Rozważmy na przykład znany program obliczający silnię:

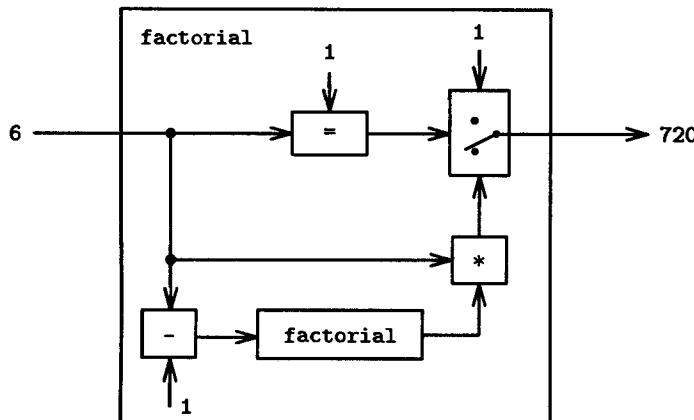
```
(define (factorial n)
  (if (= n 1)
    1
    (* (factorial (- n 1)) n)))
```

Możemy traktować ten program jako opis maszyny zawierającej części, które zmniejszają, mnożą i porównują, oraz dwupozycyjny przełącznik i kolejną maszynę obliczającą silnię. (Maszyna obliczająca silnię jest nieskończona, gdyż zawiera w sobie kolejną maszynę obliczającą silnię). Na rysunku 4.2 widać diagram przepływu dla maszyny obliczającej silnię, ukazujący połączenia między jej częściami.

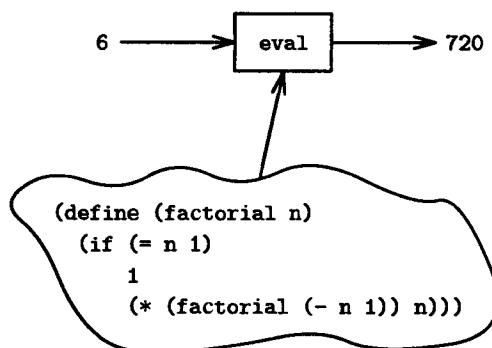
Podobnie, możemy traktować ewaluator jako bardzo specjalną maszynę, która na wejściu otrzymuje opis maszyny. Na podstawie tego opisu ewaluator konfiguruje się tak, że emuluje opisaną maszynę. Jeśli na przykład podamy ewaluatorowi definicję `factorial`, jak jest to pokazane na rys. 4.3, to ewaluator będzie mógł obliczać silnię.

Z takiego punktu widzenia nasz ewaluator stanowi *maszynę uniwersalną* (ang. *universal machine*). Naśladuje on inne maszyny, gdy są one opisane jako programy w Lispie<sup>19</sup>. Jest to frapujące. Spróbujmy wyobrazić sobie analogię.

<sup>19</sup> Fakt, że maszyny są opisane w Lispie, nie ma znaczenia. Gdybyśmy podali naszemu ewaluatorowi program w Lispie, który zachowuje się jak ewaluator jakiegoś innego języka, powiedzmy C, ewaluator Lispu emulowałby ewaluator C, który z kolei mógłby emulować dowolną maszynę opisaną jako program w C. Podobnie, pisząc ewaluator Lispu w C, otrzymamy program w C, który może wykonywać programy w Lispie. Kryje się tu głęboka myśl, że każdy ewaluator może emulować dowolny inny ewaluator. A zatem pojęcie „tego co może być w zasadzie obliczane” (pomijając takie aspekty praktyczne, jak wymagane czas i pamięć) nie zależy od języka czy komputera i odzwierciedla kryjące się tutaj pojęcie *obliczalności*. Pierwszy raz zostało to jasno pokazane przez Alana M. Turinga (1912–1954), który w artykule z 1936 r. położył podwaliny informatyki teoretycznej. W artykule tym Turing przedstawił prosty model obliczeniowy —



Rys. 4.2. Program obliczający silnię, pokazany jako maszyna abstrakcyjna



Rys. 4.3. Evaluator emulujący maszynę obliczającą silnię

giczny evaluator układów elektronicznych. Byłby to układ, który na wejściu otrzymywałby sygnał z zakodowanym schematem innego układu, takiego jak filtr. Mając taki sygnał, evaluator zacząłby się zachowywać jak filtr o takim samym opisie. Taki uniwersalny układ elektroniczny byłby prawie nie-

znany dzisiaj jako *maszyna Turinga* — i argumentował, że każdy „faktyczny proces obliczeniowy” może być sformułowany jako program dla takiej maszyny. (Argument ten jest znany jako *teza Churcha-Turinga*). Turing zaimplementował następnie maszynę uniwersalną, tzn. maszynę Turinga, która działała jak evaluator programów dla maszyn Turinga. Użył tego podejścia do pokazania, że istnieją dobrze postawione problemy, które nie mogą być obliczone za pomocą maszyny Turinga (zob. ćwiczenie 4.15), a co za tym idzie, nie mogą być sformułowane jako „faktyczne procesy obliczeniowe”. Turing poszedł dalej, a jego wkład również w informatykę praktyczną ma fundamentalny charakter. Wprowadził na przykład koncepcję programowania strukturalnego przy użyciu podprocedur ogólnego przeznaczenia. Biografia Turinga jest opisana w książce Hodgesa [52].

wyobrażalnie skomplikowany. Niezwykłe jest to, że evaluator programów jest stosunkowo prostym programem<sup>20</sup>.

Inny zadziwiający aspekt evaluatora jest taki, że stanowi on pomost między obiektami danych, którymi operuje nasz język programowania, a samym językiem programowania. Wyobraźmy sobie, że program evaluatora (zaimplementowany w Lispie) działa i że użytkownik wprowadza do niego wyrażenia i obserwuje wyniki. Z punktu widzenia użytkownika wyrażenie wejściowe takie jak `(* x x)` jest wyrażeniem w języku programowania, które evaluator powinien wykonywać. Jednak z punktu widzenia evaluatora wyrażenie to jest po prostu listą (w tym przypadku listą złożoną z trzech symboli: `*`, `x` i `x`), którą należy przetworzyć zgodnie z dobrze określonymi regułami.

To, że programy użytkowników są dla evaluatora danymi, nie powinno nas wprawiać w zakłopotanie. W rzeczywistości czasami wygodnie jest pomijać to rozróżnienie i umożliwić użytkownikowi obliczanie wprost obiektów danych jako wyrażeń lispowych poprzez udostępnienie procedury `eval` w obliczanych programach. Wiele dialektów Lispu udostępnia procedurę pierwotną `eval`, której argumentami są wyrażenie i środowisko i która oblicza dane wyrażenie w danym środowisku<sup>21</sup>. Tak więc wynikiem zarówno

```
(eval '(* 5 5) user-initial-environment)
```

jak i

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

będzie liczba 25<sup>22</sup>.

### Ćwiczenie 4.15

Mając daną jednoargumentową procedurę `p` oraz obiekt `a`, mówimy, że `p` „zatrzymuje się” dla `a`, jeżeli obliczenie wyrażenia (`p a`) kończy się przekazaniem jakiejś wartości

<sup>20</sup> Dla niektórych ludzi w sprzeczności z intuicją pozostaje fakt, że evaluator, zaimplementowany jako stosunkowo prosty program, może emulować programy, które są bardziej złożone niż on. Istnienie uniwersalnego emulatora jest głęboką i cudowną właściwością obliczeń. *Teoria funkcji rekurencyjnych*, gałąź logiki matematycznej, zajmuje się logicznymi ograniczeniami obliczeń. Douglas Hofstadter w przepięknej książce *Gödel, Escher, Bach* [53] bada niektóre z tych koncepcji.

<sup>21</sup> Uwaga: Taka procedura pierwotna `eval` różni się od procedury `eval` zaimplementowanej w punkcie 4.1.1, gdyż korzysta z rzeczywistych środowisk języka Scheme, a nie z przykładowych struktur środowiska, które zbudowaliśmy w punkcie 4.1.3. Użytkownik nie może operować na takich rzeczywistych środowiskach jak na zwykłych listach; można z nich korzystać poprzez `eval` lub inne operacje specjalne. Podobnie procedura pierwotna `apply`, którą widzieliśmy wcześniej, różni się od metacyklicznego `apply`, gdyż używa rzeczywistych procedur języka Scheme, a nie obiektów proceduralnych skonstruowanych w punktach 4.1.3 i 4.1.4.

<sup>22</sup> Implementacja MIT języka Scheme zawiera zarówno `eval`, jak i symbol `user-initial-environment`, który jest związany z początkowym środowiskiem, w którym są obliczane wyrażenia wprowadzane przez użytkownika.

(w odróżnieniu od zakończenia się z komunikatem błędu lub zapętleniem się). Pokaż, że nie da się napisać procedury `halts?`, która poprawnie określałaby, czy `p` zatrzymuje się dla `a`, dla dowolnej procedury `p` i obiektu `a`. Skorzystaj z następującego założenia: Gdyby istniała taka procedura `halts?`, można by zaimplementować taki program:

```
(define (run-forever) (run-forever))

(define (try p)
  (if (halts? p p)
      (run-forever)
      'halted))
```

Rozważ obliczenie wyrażenia `(try try)` i pokaż, że każdy możliwy wynik (zatrzymanie się lub nieskończone obliczenie) przeczy założeniu zachowania `halts?`<sup>23</sup>.

#### 4.1.6. Definicje wewnętrzne

Nasz środowiskowy model obliczeń i nasz metacykliczny evaluator wykonują definicje po kolej, rozszerzając ramkę środowiska o jedną definicję w danej chwili. Jest to szczególnie dogodne przy interakcyjnym opracowywaniu programów, kiedy to programista musi mieć możliwość dowolnego mieszania wywołań procedur z definicjami nowych procedur. Jeśli jednak przemyślimy dokładnie kwestię definicji wewnętrznych używanych do zaimplementowania struktury blokowej (wprowadzonej w punkcie 1.1.8), to okaże się, że rozszerzanie środowiska o kolejne nazwy może nie być najlepszym sposobem definiowania zmiennych lokalnych.

Rozważmy procedurę zawierającą definicje wewnętrzne, taką jak

```
(define (f x)
  (define (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
  (pozostała część treści f))
```

Naszym celem było, aby `odd?` w treści procedury `even?` odnosiło się do procedury `odd?` zdefiniowanej po procedurze `even?`. Zakresem nazwy `odd?` jest

<sup>23</sup> Chociaż przyjęliśmy, że `halts?` otrzymuje obiekt proceduralny, zauważmy, że powyższe rozumowanie można zastosować również wtedy, gdy `halts?` ma dostęp do treści procedury i jej środowiska. Jest to słynne twierdzenie Turinga o nierzozstrzygalności problemu stopu, który był pierwszym przejrzystym przykładem problemu *nieobliczalnego*, tzn. dobrze postawionego problemu, którego nie można rozwiązać w postaci procedury obliczeniowej.

cała treść `f`, a nie tylko jej część, począwszy od miejsca, w którym pojawia się `define` wprowadzające `odd?`. W rzeczywistości, gdy weźmiemy pod uwagę, że `odd?` samo jest zdefiniowane przy użyciu `even?` — a więc `even?` i `odd?` są procedurami wzajemnie rekurencyjnymi — widać, że jedyna zadowalająca interpretacja tych dwóch definicji jest taka, jakby `even?` i `odd?` były dodawane do środowiska równocześnie. Mówiąc bardziej ogólnie, w strukturze blokowej zakresem nazwy lokalnej jest cała treść procedury, w której obliczane jest `define`.

Tak się akurat składa, że nasz interpreter będzie poprawnie obliczał wywołania `f`, ale będzie się tak działało zupełnie „przez przypadek”. Ponieważ najpierw występują definicje procedur wewnętrznych, żadne ich wywołania nie będą obliczane, dopóki wszystkie one nie będą zdefiniowane. A zatem `odd?` zostanie zdefiniowane, zanim nastąpi wykonanie `even?`. W istocie dla każdej procedury, w której definicje wewnętrzne występują na początku treści, a przy obliczaniu wyrażeń definiujących wartości zmiennych nie są używane faktycznie żadne z definiowanych zmiennych, nasz sekwencyjny mechanizm obliczania będzie dawał te same wyniki co mechanizm bezpośrednio implementujący równoczesne definicje. (Przykład procedury, która nie spełnia tych ograniczeń i w przypadku której sekwencyjne obliczenie definicji nie jest równoważne równoczesnemu jej obliczeniu, można znaleźć w ćwiczeniu 4.19)<sup>24</sup>.

Istnieje jednak prosty sposób takiego traktowania definicji, że nazwy zdefiniowane wewnętrznie są wprowadzane naprawdę równocześnie — wystarczy utworzyć wszystkie zmienne lokalne, które zostaną wprowadzone do bieżącego środowiska przed obliczeniem któregokolwiek z wyrażeń mających wartość. Można tego dokonać za pomocą przekształcenia składniowego `lambda`-wyrażeń. Przed obliczeniem treści `lambda`-wyrażenia „przeglądamy” i wyłuskujemy z treści wszystkie definicje wewnętrzne. Zmienne wewnętrzne są tworzone za pomocą `let`, a następnie za pomocą przypisania ustawiane są ich wartości. Na przykład procedura

```
(lambda (parametry)
  (define u (e1))
  (define v (e2))
  (e3))
```

<sup>24</sup> To, że nie chcemy być uzależnieni od takiego mechanizmu obliczania, było powodem uwagi „Producent nie odpowiada za skutki ...” w przypisie 28, w rozdziale 1. Dzięki temu, że definicje wewnętrzne występują najpierw i nie korzystają z siebie nawzajem w trakcie obliczania definicji, standard IEEE języka Scheme pozostawia implementatorom pełną swobodę wyboru mechanizmu użytego do obliczania definicji. Wybór takiej czy innej reguły obliczania może się tutaj wydawać poboczną kwestią dotyczącą jedynie interpretacji „że napisanych” programów. W punkcie 5.5.6 zobaczymy jednak, że przyjęcie modelu równoczesnego wprowadzania definicji wewnętrznych pozwoli nam uniknąć pewnych przykrych problemów, które w przeciwnym razie miałyby miejsce przy implementowaniu kompilatora.

po takim przekształceniu miałaby postać

```
(lambda (parametry)
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (set! u (e1))
    (set! v (e2))
    (e3)))
```

gdzie `*unassigned*` jest specjalnym symbolem, który powoduje, że w przypadku próby użycia zmiennej, której nie przypisano jeszcze wartości, pobranie jej wartości spowoduje wystąpienie błędu.

Alternatywna strategia wyluskiwania definicji wewnętrznych jest przedstawiona w ćwiczeniu 4.18. W odróżnieniu od opisanego powyżej przekształcenia narzuca ona ograniczenie, że wartości definiowanych zmiennych mogą być obliczone bez używania żadnych wartości zmiennych<sup>25</sup>.

### Ćwiczenie 4.16

W tym ćwiczeniu implementujemy opisaną właśnie metodę interpretacji definicji wewnętrznych. Zakładamy, że evaluator obsługuje wyrażenia `let` (zob. ćwiczenie 4.6).

- Zmień procedurę `lookup-variable-value` (punkt 4.1.3) tak, aby zgłaszała błąd, gdy znalezioną wartością jest symbol `*unassigned*`.
- Napisz procedurę `scan-out-defines`, której argumentem jest treść procedury, a wynikiem jest równoważna treść pozbawiona definicji wewnętrznych za pomocą opisanego powyżej przekształcenia.
- Zainstaluj w interpreterze procedurę `scan-out-defines` albo w `make-procedure`, albo w `procedure-body` (zob. punkt 4.1.3). Które z tych dwóch miejsc jest lepsze? Dlaczego?

### Ćwiczenie 4.17

Narysuj diagramy środowisk powstających na skutek obliczania wyrażenia `(e3)` z przedstawionej powyżej procedury, porównując ich strukturę w przypadku interpretacji sekwencyjnej oraz opisanego wyluskiwania definicji. Dlaczego w przekształconym programie powstaje dodatkowa ramka? Wyjaśnij, dlaczego taka różnica w strukturze środowiska nigdy nie ma wpływu na działanie poprawnych programów. Zaprojektuj sposób, w jaki interpreter mógłby zaimplementować regułę „równoczesnego” wprowadzania definicji wewnętrznych bez tworzenia dodatkowej ramki.

<sup>25</sup> Standard IEEE języka Scheme dopuszcza różne strategie implementacyjne, określając, że przestrzeganie tego ograniczenia należy do programisty, a nie że implementacja powinna je narzucać. Niektóre implementacje języka Scheme, włączając w to MIT Scheme, używają powyższego przekształcenia. Tak więc niektóre programy, które nie przestrzegają opisanego ograniczenia, będą w rzeczywistości działać zgodnie z taką implementacją.

### Ćwiczenie 4.18

Rozważ alternatywną strategię wyłuskiwania definicji, która przekształca przykład z tekstu na

```
(lambda <parametry>
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (let ((a <e1>)
          (b <e2>))
      (set! u a)
      (set! v b))
    <e3>))
```

Tutaj a i b reprezentują nowe nazwy zmiennych wprowadzane przez interpreter, które nie pojawiają się w programie użytkownika. Rozważ procedurę `solve` z punktu 3.5.4:

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Czy po wyłuskaniu definicji, zgodnie z pokazaną w tym ćwiczeniu metodą, procedura ta będzie działać? A jeśli zostaną one wyłuskane zgodnie z metodą opisaną w tekście? Odpowiedź uzasadnij.

### Ćwiczenie 4.19

Ben Bajerbit, Liz P. Haker i Ewa Lu Ator spierają się o pożądany wynik obliczenia wyrażenia

```
(let ((a 1))
  (define (f x)
    (define b (+ a x))
    (define a 5)
    (+ a b))
  (f 10))
```

Ben zapewnia, że wynik powinien być uzyskany za pomocą sekwencyjnej reguły dla `define`: b jest definiowane jako 11, następnie a jest definiowane jako 5, a zatem wynik jest równy 16. Liz sprzeciwia się temu, twierdząc, że wzajemna rekursja wymaga zastosowania reguły równoczesnego wprowadzania definicji wewnętrznych procedury i że niedorzecznością jest traktowanie nazw procedur inaczej niż innych nazw. Tak więc optuje ona za mechanizmem zaimplementowanym w ćwiczeniu 4.16. W rezultacie, w momencie obliczania wartości b wartość a jest nieokreślona. A zatem w opinii Liz procedura powinna spowodować błąd. Ewa jest jeszcze innego zdania. Mówiąc, że jeżeli definicje a i b mają faktycznie być równoczesne, to w trakcie obliczania b wartość a powinna wynosić 5, b powinno być równe 15, a wynik powinien wynosić 20. Który (jeśli w ogóle którykolwiek) z przedstawionych punktów widzenia

podzielasz? Czy potrafisz opracować taki sposób implementacji definicji wewnętrznych, aby działały tak, jak chciałaby Ewa<sup>26</sup>?

### Ćwiczenie 4.20

Ponieważ definicje wewnętrzne wyglądają na sekwencyjne, podczas gdy są równoczesne, niektórzy ludzie wolą ich unikać i używać zamiast nich formy specjalnej `letrec`. `Letrec` wygląda tak jak `let`, nie jest więc zaskakujące, że zmienne, które wiążę, są wiązane równocześnie i mają wszystkie taki sam zakres. Przykładowa procedura f przedstawiona powyżej może być zapisana bez definicji lokalnych, ale przy zachowaniu jej znaczenia, jako

```
(define (f x)
  (letrec ((even?
    (lambda (n)
      (if (= n 0)
          true
          (odd? (- n 1))))))
    (odd?
      (lambda (n)
        (if (= n 0)
            false
            (even? (- n 1)))))))
  (pozostała część treści f)))
```

Wyrażenia `letrec` mają postać

```
(letrec (((v1) (e1)) ... ((vn) (en)))
  (treść))
```

i stanowią odmianę wyrażeń `let`, w których wyrażenia  $\langle e_k \rangle$  określające początkowe wartości zmiennych  $\langle v_k \rangle$  są obliczane w środowisku zawierającym wszystkie wiązania `letrec`. Umożliwia to rekursję w wiązaniach, taką jak wzajemna rekursja `even?` i `odd?` w powyższym przykładzie lub obliczenie 10 silnia za pomocą

```
(letrec ((fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1)))))))
  (fact 10)))
```

(a) Zaimplementuj `letrec` jako wyrażenie pochodne, przekształcając wyrażenia `letrec` w wyrażenia `let`, jak jest to pokazane powyżej lub w ćwiczeniu 4.18. Ozna-

<sup>26</sup> Implementatorzy MIT Scheme podzielają pogląd Liz w następującej kwestii: Ewa w zasadzie ma rację — definicje powinny być traktowane równocześnie. Jednak zaimplementowanie ogólnego, efektywnego mechanizmu, który postuluje Ewa, wydaje się trudne. Z braku takiego mechanizmu w trudnych przypadkach równoczesnych definicji lepiej jest zgłaszać błąd (pogląd Liz), niż generować niepoprawne wyniki (jak zrobiłby to Ben).

czy to, że zmienne wyrażenia `letrec` powinny być wprowadzane za pomocą `let`, a następnie za pomocą `set!` powinny im być przypisywane wartości.

(b) Ludwik Myślicielak zagubił się w tym całym zamieszaniu wokół definicji wewnętrznych. Widzi on to tak: jeśli nie lubisz używać `define` wewnątrz procedur, to możesz po prostu używać `let`. Zilustruj, co traci się przy takim rozumowaniu, rysując diagram przedstawiający środowisko, w którym obliczana jest (pozostała część treści `f`) w trakcie obliczania wyrażenia (`f 5`), gdzie `f` jest zdefiniowane tak jak w tym ćwiczeniu. Narysuj diagram środowiska dla takiego samego obliczenia, ale przy użyciu `let` zamiast `letrec` w definicji `f`.

### Ćwiczenie 4.21

Niesamowite, ale intuicja Ludwika w ćwiczeniu 4.20 jest poprawna. Faktycznie można definiować procedury rekurencyjne bez użycia `letrec` (czy nawet `define`), chociaż sposób, w jaki można to zrobić, jest dużo bardziej subtelny, niż Ludwik sobie wyobrażała. Następujące wyrażenie oblicza 10 silnia, stosując procedurę rekurencyjną<sup>27</sup>:

```
((lambda (n)
  ((lambda (fact)
    (fact fact n))
   (lambda (ft k)
     (if (= k 1)
         1
         (* k (ft ft (- k 1)))))))
 10)
```

(a) Sprawdź (obliczając wyrażenie), że to faktycznie oblicza silnię. Opracuj analogiczne wyrażenia obliczające liczby Fibonacciego.

(b) Rozważ następującą procedurę zawierającą wzajemnie rekurencyjne definicje wewnętrzne:

```
(define (f x)
  (define (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
  (even? x))
```

Uzupełnij brakujące wyrażenie w poniższej alternatywnej definicji `f` nie korzystającej ani z definicji wewnętrznych, ani z `letrec`:

<sup>27</sup> Przykład ten ilustruje sztuczkę programistyczną pozwalającą formułować procedury rekurencyjne bez używania `define`. Najbardziej ogólna sztuczka tego rodzaju polega na wprowadzeniu operatora `Y`, za pomocą którego można zaimplementować rekursję w „czystym rachunku  $\lambda$ ”. (Szczegóły rachunku lambda można znaleźć w [97], a operator `Y` jest przedstawiony w [32]).

```
(define (f x)
  ((lambda (even? odd?)
    (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? (??) (??) (??))))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? (??) (??) (??))))))
```

#### 4.1.7. Oddzielanie analizy składniowej od wykonywania

Zaimplementowany wcześniej evaluator jest prosty, ale bardzo nieefektywny, gdyż analiza składniowa wyrażeń przeplata się w nim z ich wykonywaniem. Tak więc jeśli program jest wykonywany wielokrotnie, jego składnia jest również analizowana wielokrotnie. Rozważmy na przykład obliczenie (`factorial 4`) przy użyciu następującej definicji `factorial`:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

Za każdym razem, gdy wywoływana jest procedura `factorial`, evaluator musi rozpoznać, że treść tej procedury jest wyrażeniem `if` i wydzielić z niej predykat. Dopiero wówczas może go obliczyć i działać stosownie do jego wartości. Za każdym razem, gdy obliczane jest wyrażenie `(* (factorial (- n 1)) n)` lub podwyrażenia `(factorial (- n 1))` i `(- n 1)`, evaluator musi w procedurze `eval` rozważyć odpowiednie przypadki, by określić, że wyrażenie jest kombinacją i wyłuskać operator oraz argumenty. Taka analiza jest kosztowna, a jej wielokrotne wykonywanie jest marnotrawstwem.

Możemy tak przekształcić evaluator, aby analiza składniowa była wykonywana tylko raz, dzięki czemu będzie on bardziej efektywny<sup>28</sup>. Procedurę `eval`, której argumentami są wyrażenie i środowisko, rozdzielimy na dwie części. Argumentem procedury `analyze` jest tylko wyrażenie. Dokonuje ona analizy składniowej, a jej wynikiem jest nowa procedura — *procedura wykonawcza* (ang. *execution procedure*) — zawierająca w sobie wszystko to, co ma być wykonane w ramach obliczania zanalizowanego wyrażenia. Argumentem procedury wykonawczej jest środowisko. Po jego otrzymaniu procedura ta dokonuje obliczeń. W ten sposób oszczędzamy część pracy, ponieważ dla danego wyrażenia procedura `analyze` jest wywoływana tylko raz, podczas gdy procedura wykonawcza może być wywoływana wielokrotnie.

<sup>28</sup> Technika ta stanowi integralną część procesu komplikacji, który omówimy w rozdziale 5. Jonathan Rees napisał taki interpreter języka Scheme ok. 1982 r. w ramach projektu T [86]. Niezależnie Marc Feeley wynalazł i opisał tę technikę w swojej pracy magisterskiej [24] (zob. też [25]).

Po oddzieleniu analizy od wykonywania procedura `eval` ma postać

```
(define (eval exp env)
  ((analyze exp) env))
```

Wynikiem wywołania `analyze` jest procedura wykonawcza, którą należy zastosować do środowiska. Procedura `analyze` zawiera tę samą analizę przypadków co oryginalna procedura `eval` z punktu 4.1.1, z tym że procedury obsługujące poszczególne przypadki wykonują tylko analizę, a nie pełne obliczenie

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Nieznaný rodaj wyrażenia -- ANALYZE" exp))))
```

Oto najprostsza z procedur analizy składniowej, obsługująca samoobliczające się wyrażenia. Jej wynikiem jest procedura wykonawcza, która ignoruje argument będący środowiskiem i przekazuje wyrażenie:

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))
```

W przypadku wyrażeń cytowanych możemy trochę zyskać na szybkości obliczeń, wyznaczając cytowany tekst tylko raz w trakcie analizy, a nie przy każdym wykonaniu.

```
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
```

Wyznaczanie wartości zmiennej musi nadal odbywać się w fazie wykonywania, gdyż wartość ta zależy od danego środowiska<sup>29</sup>.

---

<sup>29</sup> Jest jednak istotna część wyszukiwania wartości zmiennej, która *może* być wykonana w ramach analizy składniowej. Jak pokażemy w punkcie 5.5.6, można wyznaczyć miejsce w struk-

---

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))
```

Procedura `analyze-assignment` również musi wstrzymać ustawianie wartości zmiennej do momentu wykonania, gdy dostępne jest środowisko. Jednakże fakt, że wyrażenie `assignment-value` może być (rekurencyjnie) analizowane w fazie analizy, stanowi duży zysk w efektywności, gdyż samo wyrażenie `assignment-value` będzie teraz analizowane tylko raz. To samo dotyczy definicji.

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))
```

W przypadku wyrażeń `if` w trakcie analizy wyłuskujemy i analizujemy predykat, następnik i alternatywę.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

Analiza `lambda`-wyrażeń również daje duże polepszenie efektywności — treść `lambda`-wyrażenia jest analizowana tylko raz, mimo że procedura będąca wynikiem obliczenia `lambda`-wyrażenia może być stosowana wielokrotnie.

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

---

turze środowiska, w którym wartość zmiennej będzie się znajdować, eliminując konieczność przeglądania środowiska w poszukiwaniu danej zmiennej.

Analiza ciągu wyrażeń (takiego jak w wyrażeniu `begin` czy treści `lambda`-wyrażenia) jest zawiła<sup>30</sup>. Każde wyrażenie w ciągu jest analizowane i uzyskiwana jest jego procedura wykonawcza. Te procedury wykonawcze są łączone w jedną procedurę wykonawczą, która po otrzymaniu środowiska wywołuje po kolei wszystkie procedury wykonawcze, przekazując im otrzymane środowisko.

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Pusty ciąg -- ANALYZE")
        (loop (car procs) (cdr procs)))))
```

Analizując kombinację, analizujemy najpierw operator oraz argumenty i tworzymy procedurę wykonawczą, która wywołuje procedurę wykonawczą operatora (w celu uzyskania faktycznej procedury do zastosowania) oraz procedury wykonawcze argumentów (aby uzyskać ich faktyczne wartości). Następnie są one przekazywane do procedury `execute-application`, która jest odpowiednikiem `apply` z punktu 4.1.1. `Execute-application` różni się od `apply` tym, że treść procedury złożonej jest już zanalizowana i nie ma potrzeby jej dalszego analizowania. Zamiast tego po prostu wywołujemy procedurę wykonawczą dla treści w rozszerzonym środowisku.

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env))
                                aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                             args
                             (procedure-environment proc)))))
```

<sup>30</sup> Ćwiczenie 4.23 daje pewien wgląd w przetwarzanie ciągów wyrażeń.

```
(else
  (error
    "Nieznany rodzaj procedury -- EXECUTE-APPLICATION"
    proc))))
```

Nasz nowy evaluator używa tych samych struktur danych, procedur skądniowych i procedur wspomagających środowisko wykonania co w punktach 4.1.2, 4.1.3 i 4.1.4.

### Ćwiczenie 4.22

Rozszerz evaluator przedstawiony w niniejszym punkcie o interpretację formy spejalnej let. (Zobacz ćwiczenie 4.6).

### Ćwiczenie 4.23

Liz P. Haker nie rozumie, dlaczego procedura `analyze-sequence` musi być tak skomplikowana. Wszystkie pozostałe procedury analizujące są prostymi przekształceniami odpowiadającymi im procedur obliczających (lub klauzul procedury `eval`) z punktu 4.1.1. Oczekiwała, że `analyze-sequence` będzie miała następującą postać:

```
(define (analyze-sequence exps)
  (define (execute-sequence procs env)
    (cond ((null? (cdr procs)) ((car procs) env))
          (else ((car procs) env)
                (execute-sequence (cdr procs) env))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Pusty ciąg -- ANALYZE"))
        (lambda (env) (execute-sequence procs env))))
```

Ewa Lu Ator wyjaśnia Liz, że wersja przedstawiona w tekście wykonuje dużo większą część obliczenia ciągu w trakcie analizy. Wykonująca ciąg wyrażeń procedura Liz zamiast mieć wbudowane wywołania poszczególnych procedur wykonawczych w procedurę wykonawczą całego wyrażenia, przegląda w pętli procedury wykonawcze i wywołuje je. W rezultacie, mimo że zostały zanalizowane poszczególne wyrażenia tworzące ciąg, sam ciąg nie został zanalizowany.

Porównaj te dwie wersje `analyze-sequence`. Rozważ na przykład częsty przypadek (typowy dla treści procedur), gdy ciąg składa się tylko z jednego wyrażenia. Jakie czynności będzie wykonywać procedura wykonawcza Liz? A procedura wykonawcza przedstawiona wcześniej w tekście? Jak wypada porównanie tych dwóch wersji w przypadku ciągów złożonych z dwóch wyrażeń?

### Ćwiczenie 4.24

Zaplanuj i wykonaj kilka eksperymentów porównujących szybkość działania oryginalnego evaluatora metacyklicznego i jego wersji przedstawionej w niniejszym punkcie. Na podstawie uzyskanych wyników oszacuj dla różnych procedur, jaka część czasu przypada na analizę, a jaka na wykonanie.

## 4.2. Wariacje na temat języka Scheme — leniwe obliczanie

Teraz, gdy dysponujemy evaluatorom wyrażonym w postaci programu w Lispie, możemy eksperymentować z różnymi możliwymi decyzjami projektowymi dotyczącymi konstrukcji języka, modyfikując po prostu evaluator. W rzeczywistości nowe języki często powstają w wyniku tego, że najpierw zostaje napisany evaluator osadzający nowy język w ramach istniejącego języka wysokiego poziomu. Gdybyśmy na przykład chcieli przedyskutować z innym członkiem społeczności lisposej pewne aspekty proponowanych modyfikacji w Lispie, moglibyśmy udostępnić mu evaluator realizujący te zmiany. Mógliby on wówczas poeksperymentować z nowym evaluatorom i przekazać nam swoje uwagi oraz propozycje dalszych modyfikacji. Nie tylko użycie w implementacji języka wysokiego poziomu ułatwia testowanie i odpluskwanie evaluatora. Dodatkowo osadzenie umożliwia projektantowi podkрадanie możliwości języka, w którym evaluator jest zaimplementowany — tak jak nasz evaluator Lispu używa operacji pierwotnych i struktur sterujących z działającego pod spodem Lispu. Dopiero później (jeśli w ogóle kiedykolwiek) projektant musi zadać sobie trud skonstruowania pełnej implementacji w języku niskiego poziomu lub sprzętowej. W tym i następnym podrozdziale zbadamy pewne odmiany języka Scheme w istotny sposób zwiększące siłę wyrazu języka.

### 4.2.1. Normalna i stosowana kolejność obliczania

W podrozdziale 1.1, gdy rozpoczęliśmy rozważania na temat modeli obliczeń, stwierdziliśmy, że Scheme jest językiem o *stosowanej kolejności obliczania*, a dokładniej, że wszystkie argumenty procedury są obliczane w momencie jej wywołania. W przeciwnieństwie do tego w językach o *normalnej kolejności obliczania* obliczanie argumentów procedury jest odraczane aż do momentu, gdy faktyczne wartości argumentów są potrzebne. Odraczanie obliczania argumentów procedury do ostatniej możliwej chwili (np. aż są one potrzebne do wykonania na nich operacji pierwotnych) jest nazywane *leniwym obliczaniem* (ang. *lazy evaluation*)<sup>31</sup>. Rozważmy procedurę

```
(define (try a b)
  (if (= a 0) 1 b))
```

Obliczenie `(try 0 (/ 1 0))` powoduje w języku Scheme powstanie błędu. W przypadku leniwego obliczania nie byłoby żadnego błędu. Wynikiem obliczenia byłby 1, gdyż argument `(/ 1 0)` nigdy nie byłby obliczany.

<sup>31</sup> Różnica między określeniem „leniwe obliczanie” a „normalna kolejność obliczania” jest niezbyt jasna. Ogólnie rzecz biorąc, termin „leniwy” odnosi się do mechanizmu konkretnego evaluatora, podczas gdy „normalna kolejność” odnosi się do semantyki języka, niezależnie od konkretnej strategii obliczania. Nie jest to jednak kategoryczne rozróżnienie i te dwa terminy są często używane zamiennie.

Przykładem zastosowania leniwego obliczania może być definicja procedury `unless`:

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

której można używać w wyrażeniach w następujący sposób:

```
(unless (= b 0)
  (/ a b)
  (begin (display "wyjątek: wynik równy 0")
         0))
```

Nie będzie to działać w przypadku języka ze stosowaną kolejnością obliczania, gdyż zarówno normalna, jak i wyjątkowa wartość są obliczane przed wywołaniem `unless` (porównaj ćwiczenie 1.6). Zaletą leniwego obliczania jest to, że pewne procedury, takie jak `unless`, mogą wykonywać przydatne obliczenia, mimo że obliczenie niektórych ich argumentów może powodować błędy lub nigdy się nie kończyć.

Jeśli do treści procedury wchodzimy, zanim wartość argumentu jest obliczana, to mówimy, że procedura jest *nierzetelna* względem tego argumentu. Jeśli zaś wartość argumentu jest obliczana przed wejściem do treści procedury, to mówimy, że procedura jest *rzetelna* względem tego argumentu<sup>32</sup>. W języku o wyłącznie stosowanej kolejności obliczania wszystkie procedury są rzetelne względem swoich argumentów. W języku o wyłącznie normalnej kolejności obliczania wszystkie procedury złożone są nierzetelne względem swoich argumentów, a procedury pierwotne mogą być albo rzetelne, albo nierzetelne. Istnieją również języki (zob. ćwiczenie 4.31) umożliwiające programistom szczegółową kontrolę rzetelności definiowanych procedur.

Frapującym przykładem procedury, która w użyteczny sposób może być nierzetelna, jest `cons` (lub, w ogólności, prawie każdy konstruktor struktur danych). Można wykonywać użyteczne obliczenia, łączyć elementy w struktury danych i operować na tak powstałych strukturach danych, nawet jeśli wartości elementów nie są znane. Na przykład obliczanie długości listy ma sens, nawet jeżeli nie znamy wartości poszczególnych jej elementów. Zastosujemy tę koncepcję w punkcie 4.2.3 do zaimplementowania strumieni (przedstawionych w rozdziale 3) w postaci list nierzetelnych par.

<sup>32</sup> Określenia „rzetelna” i „nierzetelna” zasadniczo znaczą to samo, co „stosowana kolejność” i „normalna kolejność”, ale odnoszą się do poszczególnych procedur i ich argumentów, a nie do języka jako całości. Na konferencji poświęconej językom programowania można by usłyszeć, jak ktoś mówi, że „Język Hassle o normalnej kolejności obliczania zawiera pewne rzetelne konstrukcje pierwotne. Pozostałe procedury mają argumenty obliczane leniwie”.

### Ćwiczenie 4.25

Przypuśćmy, że (w języku Scheme ze zwykłą stosowaną kolejnością obliczania) zdefiniujemy procedurę `unless` tak, jak to miało miejsce powyżej, a następnie przy jej użyciu zdefiniujemy w następujący sposób procedurę `factorial`:

```
(define (factorial n)
  (unless (= n 1)
    (* n (factorial (- n 1)))
    1))
```

Co się stanie, jeśli spróbujemy obliczyć `(factorial 5)`? Czy taka definicja zadziałałaby w języku z normalną kolejnością obliczania?

### Ćwiczenie 4.26

Ben Bajerbit i Liz P. Haker nie zgadzają się ze sobą co do roli leniwego obliczania w implementowaniu takich rzeczy jak `unless`. Ben zwraca uwagę, że przy stosowanej kolejności obliczania można zaimplementować `unless` jako formę specjalną. Liz odparowuje ten argument, mówiąc, że gdyby ktoś tak zrobił, to `unless` byłoby jedynie formą składniową, a nie procedurą, której można użyć w połączeniu z procedurami wyższych rzędów. Dopowiedź szczegóły argumentacji obu stron. Pokaż, jak można zaimplementować `unless` jako wyrażenie pochodne (tak jak `cond` bądź `let`) oraz podaj przykład sytuacji, w której mogłoby się przydać, żeby `unless` było dostępne jako procedura, a nie forma specjalna.

#### 4.2.2. Interpreter z leniwym obliczaniem

W niniejszym punkcie zaimplementujemy język o normalnej kolejności obliczania, który jest taki sam jak Scheme z wyjątkiem tego, że wszystkie procedury złożone są nierzetelne wobec swoich argumentów. Procedury pierwotne będą nadal rzetelne. Nie jest trudno zmodyfikować evaluatora z punktu 4.1.1 tak, aby interpretowany przez niego język zachowywał się w ten sposób. Prawie wszystkie wymagane zmiany koncentrują się wokół stosowania procedur.

Podstawowa koncepcja polega na tym, aby w momencie stosowania procedury interpreter określił, które argumenty mają być obliczone, a które mają być odroczone. Argumenty odroczone nie są obliczane; zamiast tego są przekształcane w swoje domknięcia (ang. *thunks*)<sup>33</sup>. Domknięcie musi zawierać informacje wymagane do uzyskania wartości argumentu, gdy jest ona potrzebna, tak jakby była obliczana w momencie stosowania procedury. Tak więc

<sup>33</sup> Angielskie słowo *thunk* zostało wymyślone przez nieformalną grupę roboczą, która dyskutowała na temat implementacji przekazywania argumentów przez nazwę (ang. *call-by-name*) w Algol 60. Zauważali oni, że większa część analizy związanej z „przemyśleniem” (ang. „thinking about”) wyrażeń może być wykonana w trakcie kompilacji; tak więc w trakcie wykonywania programu wyrażenie „przemyślanego” („thunk”) [56]. [Jak podają twórcy tego określenia, „thunk” to czas przeszły od «think» o drugiej nad ranem”; przyp. tłum.].

domknięcie musi zawierać wyrażenie argumentu oraz środowisko, w którym zastosowanie procedury jest obliczane.

Proces obliczania domknięcia wyrażenia jest nazywany *wymuszaniem* (ang. *forcing*)<sup>34</sup>. Zwykle obliczenie wartości domknięcia będzie wymuszane tylko wtedy, kiedy ta wartość będzie potrzebna: gdy będzie ona przekazywana procedurze pierwotnej, która używa tej wartości; gdy będzie ona wartością predykatu instrukcji warunkowej; gdy będzie ona wartością operatora, który ma być właśnie zastosowany jako procedura. Jedna z decyzji projektowych, jakie musimy podjąć, dotyczy tego, czy *spamiętywać*, czy też nie, wartości domknąć, tak jak to było z obiektami odroczonymi w punkcie 3.5.1. W przypadku spamiętywania, gdy po raz pierwszy wymuszamy obliczenie domknięcia, zapamiętana jest obliczona wartość. Przy kolejnych wymuszeniach przekazywana jest po prostu zapamiętowana wartość bez powtarzania obliczeń. Nasz interpreter będzie spamiętywał, gdyż w przypadku wielu zastosowań jest to bardziej efektywne. Jest tu jednak kilka ślepskich kwestii do rozważenia<sup>35</sup>.

### Modyfikowanie evaluatora

Główna różnica między leniwym evaluatorem a tym z podrozdziału 4.1 polega na obsłudze stosowania procedur w `eval` i `apply`.

Klauzula `application?` procedury `eval` przyjmuje postać

```
((application? exp)
  (apply (actual-value (operator exp) env)
         (operands exp)
         env))
```

Jest ona prawie taka sama jak klauzula `application?` procedury `eval` z punktu 4.1.1. Jednak w przypadku leniwego obliczania wywołujemy `apply`, przekazując wyrażenia argumentów, a nie wartości uzyskane na skutek ich obliczenia. Jeżeli argumenty mają być odroczone, to do konstrukcji domknąć będziemy

<sup>34</sup> Jest to odpowiednik zastosowania `force` do obiektów odroczonych, które wprowadziliśmy w rozdziale 3 w celu reprezentowania strumieni. Zasadnicza różnica między tym, co robimy teraz, a tym, co robiliśmy w rozdziale 3, polega na tym, że teraz umieszczamy odraczanie i wymuszanie w evaluatorze, tym samym wprowadzając te mechanizmy do języka w automatyczny i jednolity sposób.

<sup>35</sup> Leniwe obliczanie w połączeniu ze spamiętywaniem jest czasem określane jako przekazywanie argumentów obliczanych na żądanie (ang. *call-by-need*) dla odróżnienia od przekazywania argumentów przez nazwę (ang. *call-by-name*). (Przekazywanie argumentów przez nazwę, wprowadzone w Algolu 60, jest podobne do leniwego obliczania bez spamiętywania). Jako konstruktorzy języka możemy wybrać, czy evaluator ma spamiętywać, czy nie spamiętywać, bądź zostawić to jako opcję do wyboru programistów (ćwiczenie 4.31). Jak można się spodziewać, wybór ten w powiązaniu z przypisaniami wiąże się z pewnymi subtelnymi i pogmatwanymi kwestiami. (Zobacz ćwiczenia 4.27 i 4.29). W doskonałym artykule [13] Clinger stara się wyjaśnić wielorakie wymiary narosłych tutaj niejasności.

potrzebować środowiska, a więc musimy je również przekazywać. Nadal obliczamy wartość operatora, ponieważ `apply` potrzebuje faktycznej procedury, która ma być zastosowana, aby móc rozróżnić jej rodzaj (pierwotna czy złożona) i zastosować ją.

Za każdym razem, gdy potrzebna jest nam faktyczna wartość wyrażenia, używamy procedury

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

zamiast tylko `eval`, tak więc jeżeli wartością wyrażenia jest domknięcie, to zostanie wymuszone obliczenie jego wartości.

Nasza nowa wersja `apply` jest prawie taka sama jak wersja z punktu 4.1.1. Różnica polega na tym, że `eval` przekazuje nieobliczone wyrażenia argumentów — w przypadku procedur pierwotnych (które są rzetelne), przed zastosowaniem procedury pierwotnej obliczamy wartości wszystkich argumentów; w przypadku procedur złożonych (które nie są rzetelne), przed zastosowaniem procedury odraczamy wszystkie argumenty.

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env))) ; zmienione
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ; zmienione
           (procedure-environment procedure))))
        (else
         (error
          "Nieznaný rodaj procedury -- APPLY" procedure))))
```

Procedury przetwarzające argumenty są dokładnie takie jak `list-of-values` z punktu 4.1.1 z wyjątkiem tego, że `list-of-delayed-args` odracza argumenty, zamiast obliczać ich wartości, a `list-of-arg-values` używa `actual-value` zamiast `eval`:

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                               env))))
```

```
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
    '()
    (cons (delay-it (first-operand exps) env)
          (list-of-delayed-args (rest-operands exps)
                                env))))
```

Kolejne miejsce, w którym musimy zmienić evaluator, to obsługa wyrażenia `if`, gdzie zamiast `eval` musimy użyć `actual-value`, aby uzyskać wartość wyrażenia predykatu, zanim zbadamy, czy jest to prawda, czy fałsz:

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
    (eval (if-consequent exp) env)
    (eval (if-alternative exp) env)))
```

Na koniec musimy zmienić procedurę `driver-loop` (punkt 4.1.4) w ten sposób, żeby zamiast `eval` używała `actual-value`, aby w przypadku gdy na wyjściu z pętli wczytaj-oblicz-wypisz uzyskamy odroczoną wartość, wymusić jej obliczenie przed wypisaniem. Zmieniamy również znaki zachęty i systemu, aby było widać, że mamy do czynienia z leniwym evaluatorom:

```
(define input-prompt ";;; L-Eval wprowadź wyrażenie:")
(define output-prompt ";;; L-Eval wartość:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
          (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

Po dokonaniu tych zmian możemy uruchomić evaluator i przetestować go. Udane obliczenie omówionego w punkcie 4.2.1 wyrażenia `try` wskazuje, że evaluator wykonuje leniwe obliczenia:

```
(define the-global-environment (setup-environment))

(driver-loop)

;;; L-Eval wprowadź wyrażenie:
(define (try a b)
  (if (= a 0) 1 b))
;;; L-Eval wartość:
ok
```

```
;;; L-Eval wprowadź wyrażenie:
(try 0 (/ 1 0))
;;; L-Eval wartość:
1
```

### Reprezentowanie domknięć

Nasz evaluator musi zadbać o to, aby w momencie stosowania procedur do argumentów były tworzone ich domknięcia i aby później wymuszane było obliczanie ich wartości. Domknięcie musi zawierać w sobie wyrażenie razem ze środowiskiem, ażeby można było później obliczyć wartość argumentu. Chcąc wymusić obliczenie wartości domknięcia, wydobywamy z niego wyrażenie i środowisko i obliczamy wartość danego wyrażenia w danym środowisku. Zamiast eval używamy `actual-value`, aby w przypadku gdy wartość wyrażenia sama jest domknięciem, wymusić obliczenie jego wartości — itd. aż do uzyskania czegoś, co nie jest domknięciem:

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

Jeden z prostych sposobów na zapamiętanie razem wyrażenia i środowiska polega na utworzeniu listy zawierającej to wyrażenie i środowisko. Tak więc domknięcia tworzymy w następujący sposób:

```
(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))
```

W rzeczywistości nie jest to dokładnie to, czego potrzebujemy w naszym interpreterze — potrzebne nam są raczej domknięcia ze spamiętywaniem. Gdy wymuszamy obliczenie wartości domknięcia, zmieniamy je w obliczone domknięcie, zastępując pamiętane wyrażenie jego wartością i zmieniając znacznik `thunk`, tak aby było wiadomo, że obliczono wartość tego domknięcia<sup>36</sup>.

---

<sup>36</sup> Zwróćmy uwagę, że w momencie gdy wartość wyrażenia jest już obliczona, usuwamy również z domknięcia środowisko `env`. Nie ma to wpływu na wynik obliczany przez interpreter. Pomaga jednak oszczędzać pamięć, ponieważ usunięcie z domknięcia odniesienia (referencji) do `env`, gdy środowisko to już nie jest potrzebne, pozwala na usunięcie go przez mechanizm *odśmiecania*, który omówimy w podrozdziale 5.3, i ponowne użycie zajmowanej przez nie pamięci.

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                         (thunk-exp obj)
                         (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result) ; zastąp exp jego wartością
           (set-cdr! (cdr obj) '()) ; zapomnij niepotrzebne env
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

Zauważmy, że ta sama procedura `delay-it` działa zarówno ze spamiętywaniem, jak i bez niego.

### Ćwiczenie 4.27

Przypuśćmy, że wprowadzamy do leniwego ewaluatora następujące definicje:

```
(define count 0)

(define (id x)
  (set! count (+ count 1))
  x)
```

Podaj i wyjaśnij brakujące odpowiedzi w następującym ciągu interakcji<sup>37</sup>:

```
(define w (id (id 10)))

;;; L-Eval wprowadź wyrażenie:
count
;;; L-Eval wartość:
<odpowiedź>

;;; L-Eval wprowadź wyrażenie:
w
;;; L-Eval wartość:
<odpowiedź>
```

Podobnie moglibyśmy usuwać niepotrzebne środowiska z odroczeniowych obiektów ze spamiętywaniem z punktu 3.5.1, sprawiając by procedura `memo-proc` po obliczeniu wartości obiektu robiła coś w rodzaju (`set! proc '()`) w celu pozbycia się procedury `proc` (zawierającej w sobie środowisko, w którym obliczono `delay`), umożliwiając tym samym usunięcie tego środowiska przez mechanizm odśmiecania.

<sup>37</sup> Ćwiczenie to pokazuje, że wzajemne oddziaływanie między leniwym obliczaniem a efektami ubocznymi mogą być bardzo mylące. Jest to dokładnie to, czego można się spodziewać na podstawie rozważań z rozdziału 3.

```
;;; L-Eval wprowadź wyrażenie:
count
;;; L-Eval wartość:
<odpowiedź>
```

### Ćwiczenie 4.28

W procedurze eval do obliczenia operatora, przed przekazaniem go do apply, zamiast eval używamy actual-value w celu wymuszenia obliczenia jego wartości. Podaj przykład pokazujący, że takie wymuszenie jest konieczne.

### Ćwiczenie 4.29

Podaj przykład programu, który według Ciebie działałby bez spamiętywania dużo wolniej niż ze spamiętywaniem. Rozważ również następującą interakcję, w której procedura id jest zdefiniowana tak jak w ćwiczeniu 4.27, a count jest początkowo równe 0:

```
(define (square x)
  (* x x))

;;; L-Eval wprowadź wyrażenie:
(square (id 10))
;;; L-Eval wartość:
<odpowiedź>

;;; L-Eval wprowadź wyrażenie:
count
;;; L-Eval wartość:
<odpowiedź>
```

Podaj brakujące odpowiedzi dla ewaluatora ze spamiętywaniem i bez niego.

### Ćwiczenie 4.30

Efek Tuboczny, nawrócony programista w C, martwi się, że pewne efekty uboczne mogą nigdy nie mieć miejsca, gdyż leniwy ewaluator nie wymusza obliczania wyrażeń tworzących ciągi. Skoro wartości wyrażeń tworzących ciąg, z wyjątkiem ostatniego z nich, nie są używane (wyrażenia te istnieją ze względu na swoje efekty uboczne, takie jak przypisania do zmiennych czy wypisywanie), potrzeba obliczenia ich wartości (taka jak np. przekazanie jako argument do procedury pierwotnej), w następstwie której wymuszone byłoby obliczenie ich wartości, może nigdy się nie pojawić. Tak więc Efek uważa, że w trakcie obliczania ciągu wyrażeń należy wymusić obliczenie wszystkich wyrażeń w ciągu z wyjątkiem ostatniego. Proponuje on, żeby tak zmodyfikować procedurę eval-sequence z punktu 4.1.1, aby zamiast eval używała actual-value:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (actual-value (first-exp exps) env)
              (eval-sequence (rest-exp exps) env))))
```

(a) Ben Bajerbit uważa, że Efek nie ma racji. Pokazuje on Efkiowi procedurę `for-each` opisaną w ćwiczeniu 2.23, która jest ważnym przykładem ciągu wyrażeń z efektami ubocznymi:

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
              (for-each proc (cdr items)))))
```

Twierdzi on, że ewaluator opisany w tekście (z oryginalną procedurą `eval-sequence`) wykona ją poprawnie:

```
;;; L-Eval wprowadź wyrażenie:
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
```

```
57
321
88
;;; L-Eval wartość:
done
```

Wyjaśnij, dlaczego Ben ma rację co do działania `for-each`.

(b) Efek zgadza się, że Ben ma rację z przykładem `for-each`, ale mówi, że nie jest to ten typ programów, o których myślał, proponując zmianę `eval-sequence`. Zdefiniował on w leniwym ewaluatorze następujące dwie procedury:

```
(define (p1 x)
  (set! x (cons x '(2)))
  x)

(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))
```

Jakie są wyniki `(p1 1)` i `(p2 1)` przy oryginalnej procedurze `eval-sequence`? Jakie byłyby ich wartości po wprowadzeniu do `eval-sequence` zmian proponowanych przez Efka?

(c) Efek wskazuje, że proponowane przez niego zmiany w `eval-sequence` nie wpływają na działanie przykładu z części (a). Wyjaśnij, dlaczego faktycznie tak jest.

(d) Jak sądzisz, jak należy w leniwym ewaluatorze traktować ciągi wyrażeń? Czy podoba Ci się podejście Efka, podejście opisane w tekście, czy też jeszcze jakieś inne rozwiązanie?

### Ćwiczenie 4.31

Przedstawione w niniejszym podrozdziale podejście jest trochę nieprzyjemne, gdyż wprowadza do języka Scheme zmiany niezgodne z przyjętymi ustaleniami. Byłoby miło, gdyby zaimplementowano leniwe obliczanie jako *rozszerzenie wstecznie kompatybilne*, tzn. takie, że zwykłe programy w języku Scheme działałyby tak jak dotychczas. Możemy tego dokonać, rozszerzając składnię deklaracji procedur i pozwalając użytkownikowi decydować, czy argumenty mają być odraczane. Przy okazji możemy

również pozwolić użytkownikowi wybierać między odraczaniem ze spamiętywaniem i bez. Na przykład definicja

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

definiowałaby `f` jako czteroargumentową procedurę, której pierwszy i trzeci argument są obliczane w momencie wywołania procedury, drugi argument jest odraczany, a trzeci argument jest zarówno odraczany, jak i spamiętywany. A zatem zwykłe definicje procedur działałyby tak samo jak w zwykłym języku Scheme, natomiast dodawanie do każdego parametru każdej procedury złożonej deklaracji `lazy-memo` powodowałyby leniwe działanie evaluatora, tak jak to zostało zdefiniowane w niniejszym podrozdziale. Zaprojektuj i zaimplementuj zmiany wymagane do wprowadzenia takiego rozszerzenia języka Scheme. Będziesz musiał zaimplementować nowe procedury składniowe obsługujące nową składnię `define`. Musisz również zadbać, aby `eval` lub `apply` określały, kiedy argumenty należy odroczyć, i stosownie do tego wymuszać lub odracać obliczanie ich wartości; a także musisz zadbać o to, aby przy wymuszaniu odpowiednio spamiętywać, lub nie, obliczane wartości.

#### 4.2.3. Strumienie jako listy leniwe

W punkcie 3.5.1 pokazaliśmy, jak można zaimplementować strumienie jako listy odroczone. Wprowadziliśmy formy specjalne `delay` i `cons-stream`, które umożliwiały nam skonstruowanie „obietnicy” obliczenia `cdr` strumienia, bez konieczności faktycznego spełnienia tej obietnicy (aż do pewnego czasu). Moglibyśmy używać tej ogólnej techniki wprowadzania form specjalnych za każdym razem, gdy potrzebujemy większej kontroli nad procesem obliczania, ale byłoby to niewygodne. Przede wszystkim formy specjalne nie są obiektami pierwszorzędnymi, takimi jak procedury, więc nie możemy używać ich w połączeniu z procedurami wyższych rzędów<sup>38</sup>. Ponadto jesteśmy zmuszeni budować strumienie jako nowy rodzaj obiektów danych, podobny, ale nie identyczny z listami, co wymaga ponownego zaimplementowania wielu zwykłych operacji na listach (`map`, `append` itd.) dla strumieni.

Przy leniwym obliczaniu strumienie i listy mogą być identyczne, dzięki czemu nie będą potrzebne żadne dodatkowe formy specjalne ani rozdzielenie operacji na listach i na strumieniach. Musimy jedynie zadbać o to, aby `cons` był nierzetelny. Można tego dokonać, rozszerzając leniwy evaluator tak, aby dopuszczał nierzetelne operacje pierwotne, i zaimplementować `cons` jako jedną z nich. Prościej jednak jest przypomnieć sobie (punkt 2.1.3), że nie ma żadnego zasadniczego powodu, aby `cons` był zaimplementowany jako operacja pierwotna. Zamiast tego możemy reprezentować pary jako procedury<sup>39</sup>:

<sup>38</sup> Jest to dokładnie kwestia dotycząca procedury `unless`, przedstawiona w ćwiczeniu 4.26.

<sup>39</sup> Jest to reprezentacja proceduralna opisana w ćwiczeniu 2.4. Zasadniczo dowolna reprezentacja proceduralna (np. implementacja polegająca na przekazywaniu komunikatów) nadawałaby

---

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
```

Za pomocą tych podstawowych operacji standardowe definicje operacji na listach będą działały zarówno dla skończonych, jak i dla nieskończonych list (strumieni), a operacje na strumieniach mogą być zaimplementowane jako operacje na listach. Oto kilka przykładów:

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1)))))

(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))

(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))

(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))

(define ones (cons 1 ones))

(define integers (cons 1 (add-lists ones integers)))

;; L-Eval wprowadź wyrażenie:
(list-ref integers 17)
;; L-Eval wartość:
18
```

---

się równie dobrze. Zwrócmy uwagę, że możemy zainstalować te definicje w leniwym evaluatorze, po prostu wprowadzając je w trakcie działania pętli sterującej. Gdybyśmy na początku wprowadzili do środowiska globalnego `cons`, `car` i `cdr` jako operacje pierwotne, to byłyby one przeddefiniowane. (Zobacz również ćwiczenia 4.33 i 4.34).

Zauważmy, że te listy leniwe są jeszcze bardziej leniwe niż strumienie z rozdziału 3 — zarówno `car`, jak i `cdr` listy jest odroczony<sup>40</sup>. W rzeczywistości nawet dostęp do `car` lub `cdr` pary leniwej nie musi wymuszać obliczenia wartości elementu listy. Obliczenie wartości zostanie wymuszone tylko wtedy, gdy faktycznie będzie potrzebne — na przykład do użycia jako argument operacji pierwotnej lub do wypisania jako wynik.

Pary leniwe mogą również rozwiązać problem dotyczący strumieni, który pojawił się w punkcie 3.5.4, gdzie okazało się, że formułowanie modeli strumieniowych systemów zawierających pętle może wymagać umieszczenia w programie jawnych operacji `delay` poza tymi, które są wprowadzane przez `cons-stream`. W przypadku leniwego obliczania wszystkie argumenty procedur są odraczane w jednolity sposób. Możemy na przykład zaimplementować procedury, które całkują listy i rozwiązują równania różniczkowe, tak jak to pierwotnie zamierzaliśmy w punkcie 3.5.4:

```
(define (integral integrand initial-value dt)
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt)
                     int)))
  int)

(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)

;;; L-Eval wprowadź wyrażenie:
(list-ref (solve (lambda (x) x) 1 0.001) 1000)
;;; L-Eval wartość:
2.716924
```

### Ćwiczenie 4.32

Podaj kilka przykładów ilustrujących różnicę między strumieniami z rozdziału 3 i „bardziej leniwymi” listami leniwyimi opisanymi w niniejszym punkcie. Jak można zastosować tę dodatkową leniwość?

### Ćwiczenie 4.33

Ben Bajerbit testuje powyższą implementację list leniwyh, obliczając wyrażenie

```
(car '(a b c))
```

Ku jego zaskoczeniu powoduje to błąd. Po chwili namysłu zdaje sobie sprawę, że „listy” powstałe przez wczytywanie cytowanych wyrażeń różnią się od list, którymi

<sup>40</sup> Pozwala to na tworzenie odroczonych wersji bardziej ogólnych rodzajów struktur listowych, a nie tylko ciągów. Hughes [54] omawia pewne zastosowania „drzew leniwyh”.

operują nowe definicje `cons`, `car` i `cdr`. Zmodyfikuj sposób, w jaki evaluator traktuje cytowane wyrażenia, ażeby cytowana lista wprowadzona w trakcie działania pętli sterującej była faktycznie listą leniową.

### Ćwiczenie 4.34

Zmodyfikuj pętlę sterującą evaluatora tak, aby leniwe pary i listy były wypisywane w jakiś sensowny sposób. (Co zrobisz z listami nieskończonymi?) Być może będziesz musiał tak zmodyfikować reprezentację leniwych par, aby evaluator mógł rozpoznać je w celu ich wypisania.

## 4.3. Wariacje na temat języka Scheme — obliczenia niedeterministyczne

W niniejszym podrozdziale rozszerzamy evaluator języka Scheme, aby działał zgodnie z paradygmatem programowania nazywanym *obliczeniami niedeterministycznymi*, wbudowując w evaluator mechanizm automatycznego przeszukiwania. Jest to dużo głębsza zmiana w języku niż wprowadzenie leniwego obliczania w podrozdziale 4.2.

Obliczenia niedeterministyczne, tak jak przetwarzanie strumieni, są przydatne w przypadku zastosowań typu „wygeneruj i sprawdź”. Rozważmy zadanie polegające na przejrzeniu dwóch list liczb całkowitych i znalezieniu pary takich liczb — jednej z pierwszej listy, a drugiej z drugiej listy — których suma jest liczbą pierwszą. Widzieliśmy już, jak można sobie z tym poradzić za pomocą operacji na ciągach skończonych (punkt 2.2.3) oraz za pomocą strumieni nieskończonych (punkt 3.5.3). Nasze podejście polegało na wygenerowaniu ciągu wszystkich możliwych par i odfiltrowaniu par, których sumy są liczbami pierwszymi. To, czy faktycznie generujemy najpierw cały ciąg par, jak w rozdziale 2, czy też przeplatamy generowanie i filtrowanie, jak w rozdziale 3, jest nieistotne dla zasadniczej koncepcji organizacji obliczeń.

Podejście niedeterministyczne wprowadza inną koncepcję. Wyobraźmy sobie, że po prostu wybieramy (w jakiś sposób) liczbę z pierwszej listy i liczbę z drugiej listy, wymagając przy tym (za pomocą pewnego mechanizmu), żeby ich suma była liczbą pierwszą. Wyraża to następująca procedura:

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

Może się wydawać, jakby ta procedura była jedynie innym sformułowaniem problemu, a nie określeniem sposobu jego rozwiązania. Niemniej jednak jest to poprawny program niedeterministyczny<sup>41</sup>.

<sup>41</sup> Zakładamy, że wcześniej zdefiniowano procedurę `prime?`, która sprawdza, czy dana liczba jest pierwsza. Jednak nawet mając zdefiniowane `prime?`, procedura `prime-sum-pair` mo-

Kluczowy pomysł polega na tym, że w niedeterministycznym języku wyrażenia mogą mieć więcej niż jedną możliwą wartość. Na przykład wartością `an-element-of` może być dowolny element danej listy. Nasz ewaluator programów niedeterministycznych będzie automatycznie wybierał ewentualną wartość, śledząc dopuszczalne możliwości. Jeśli kolejne wymaganie nie jest spełnione, ewaluator spróbuje sprawdzić inną możliwość i będzie tak próbował, aż obliczenie się powiedzie lub wszystkie możliwości zostaną rozpatrzone. Tak jak leniwy ewaluator uwalniał programistę od konieczności śledzenia szczegółów tego, kiedy obliczanie wartości jest odraczane, a kiedy wymuszane, tak ewaluator programów niedeterministycznych uwalnia programistę od konieczności śledzenia szczegółów tego, jak dokonywane są wybory.

Pouczające jest porównanie koncepcji dotyczących czasu wprowadzonych przez obliczenia niedeterministyczne oraz przetwarzanie strumieni. Przetwarzanie strumieni używa leniwego obliczania do oddzielenia momentów, w których budowany jest strumień możliwych wyników, od momentów, w których elementy strumienia są faktycznie generowane. Ewaluator podtrzymuje złudzenie, że wszystkie możliwe wyniki przedstawiono nam w formie ponadczasowego ciągu. W przypadku obliczeń niedeterministycznych wyrażenie reprezentuje badanie zbioru możliwych światów, z których każdy jest określony przez wiele wyborów. Niektóre z możliwych światów prowadzą w ślepe zaułki, podczas gdy inne niosą ze sobą przydatne wartości. Ewaluator programów niedeterministycznych podtrzymuje złudzenie, że czas się rozgałęzia i że nasze programy mają wiele różnych możliwych historii wykonania. Gdy dochodzimy do ślepego zaułka, możemy cofnąć się do miejsca dokonania ostatniego wyboru i wybrać inną gałąź czasu.

Zaimplementowany poniżej ewaluator programów niedeterministycznych jest nazywany ewaluatorem `amb`, gdyż opiera się na nowej formie specjalnej nazywanej `amb`. Możemy wprowadzić do ewaluatora `amb` definicję `prime-sum-pair` (wraz z definicjami `prime?`, `an-element-of` i `require`) i uruchomić tę procedurę w następujący sposób:

```
;;; Amb-Eval wprowadź wyrażenie:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Rozpoczęcie rozwiązywania nowego problemu
;;; Amb-Eval wartość:
(3 20)
```

---

że wyglądać podejrzanie, podobnie jak nieudana próba zdefiniowania funkcji pierwiastkującej w „pseudo-Lispie”, którą przedstawiliśmy na początku punktu 1.1.7. W rzeczywistości przy takim podejściu procedura pierwiastkowania może być sformułowana jako program niedeterministyczny. Włączając do ewaluatora mechanizm przeszukiwania, zacieramy rozróżnienie między opisami całkowicie deklaratywnymi oraz imperatywnymi określeniami sposobów obliczania wyników. Pójdziemy dalej w tym kierunku w podrozdziale 4.4.

Wartość będąca wynikiem została uzyskana po wielokrotnym wybieraniu elementów z list, aż został dokonany trafny wybór.

W punkcie 4.3.1 wprowadzamy formę `amb` i wyjaśniamy, w jaki sposób realizuje ona niedeterminizm, korzystając z automatycznego mechanizmu przeszukiwania wbudowanego w evaluator. W punkcie 4.3.2 przedstawiamy przykłady programów niedeterministycznych, a w punkcie 4.3.3 podajemy szczegóły tego, jak modyfikując zwykły evaluator języka Scheme, można zaimplementować evaluator `amb`.

### 4.3.1. Amb i przeszukiwanie

Aby rozszerzyć język Scheme o niedeterminizm, wprowadzamy nową formę specjalną `amb`<sup>42</sup>. Wartością wyrażenia  $(\text{amb } \langle e_1 \rangle \langle e_2 \rangle \dots \langle e_n \rangle)$  jest wartość jednego z  $n$  wyrażeń  $\langle e_i \rangle$ , wybrana „niejednoznacznie”. Na przykład wyrażenie

```
(list (amb 1 2 3) (amb 'a 'b))
```

może mieć sześć możliwych wartości:

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

Jeśli `amb` ma tylko jedną możliwość do wyboru, to jej wynikiem jest zwykła (jedna) wartość.

Amb bez żadnych możliwości wyboru — wyrażenie `(amb)` — to wyrażenie pozbawione akceptowalnych wartości. Z operacyjnego punktu widzenia możemy myśleć o `(amb)` jak o wyrażeniu, które powoduje, że obliczenie kończy się „niepowodzeniem” — obliczenie jest przerywane i żadna wartość nie jest przekazywana jako wynik. Stosując takie podejście, możemy w następujący sposób wyrazić wymaganie, że określony predykat `p` musi być spełniony:

```
(define (require p)
  (if (not p) (amb)))
```

Za pomocą `amb` i `require` możemy tak zaimplementować użytą wcześniej procedurę `an-element-of`:

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

Jeśli lista elementów jest pusta, to obliczenie `an-element-of` kończy się niepowodzeniem. W przeciwnym razie wynik jest określony niejednoznacznie

<sup>42</sup> Pomysł wyrażania programów niedeterministycznych za pomocą formy `amb` został po raz pierwszy opisany w 1961 r. przez Johna McCarthy'ego [72].

i jest równy pierwszemu elementowi listy lub elementowi wybranemu spośród pozostałych elementów.

Możemy również wyrażać wybory dokonywane spośród nieskończonej liczby możliwości. Wynikiem następującej procedury może być potencjalnie dowolna liczba całkowita większa lub równa danej liczbie  $n^*$ :

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

Przypomina to procedurę `integers-starting-from` generującą strumień, opisaną w punkcie 3.5.2, z tą jednak różnicą, że wynikiem `integers-starting-from` jest obiekt reprezentujący ciąg wszystkich liczb całkowitych, począwszy od  $n$ , a wynikiem procedury `an-integer-starting-from` jest jedna liczba całkowita<sup>43</sup>.

Możemy sobie abstrakcyjnie wyobrazić, że obliczanie wyrażenia `amb` powoduje rozgałęzienie się czasu i dla każdej z możliwych wartości wyrażenia obliczenia biegną wzdłuż osobnej gałęzi. Mówimy, że `amb` reprezentuje *wybór niedeterministyczny*. Gdybyśmy dysponowali maszyną o wystarczającej liczbie procesorów, które mogłyby być dynamicznie przydzielane, moglibyśmy w prosty sposób zaimplementować przeszukiwanie. Wykonywanie przebiegałoby tak jak w maszynie sekwencyjnej aż do napotkania wyrażenia `amb`. W momencie napotkania takiego wyrażenia przydzielane by były dodatkowe procesory, tak aby dla każdego z możliwych wyborów jeden z procesorów kontynuował sekwencyjne wykonanie programu dla tego wyboru. Każdy procesor działałby sekwencyjnie, tak jakby wykonywał jedyny możliwy wybór, aż do udanego zakończenia obliczeń, zakończenia ich niepowodzeniem lub też obliczenie dzieliłoby się dalej<sup>44</sup>.

\* Istnieje subtelna różnica między dokonywaniem wyboru spośród skończonej i nieskończonej liczby możliwości. Jeżeli używamy opisanej formy `amb` i zbiór możliwych wartości jest nieskończony, to nie możemy mieć pewności, że obliczenie wyrażenia się zakończy; zob. E. W. Dijkstra: *Umiejętność programowania*, WNT, Warszawa 1985, rozdz. 9 (przyp. tłum.).

<sup>43</sup> W rzeczywistości rozróżnienie między przekazaniem jednej wybranej niedeterministycznie możliwości a przekazaniem wszystkich możliwości zależy w pewnym stopniu od punktu widzenia. Z punktu widzenia kodu używającego tej wartości wynikiem niedeterministycznego wyboru jest jedna wartość. Z punktu widzenia programisty konstruującego program wynikiem niedeterministycznego wyboru są potencjalnie wszystkie możliwe wartości i obliczenie się rozgałęzia, tak że każda z nich jest badana z osobna.

<sup>44</sup> Można by się sprzeczać, że taki mechanizm jest beznadziejnie nieefektywny. Przy takim podejściu rozwiązywanie prosto postawionego problemu może wymagać zastosowania milionów procesorów, a większość z nich przez większość czasu byłaby bezczynna. Taką argumentację należy rozważyć w kontekście historycznym. Swojego czasu pamięć była uważana za drogi towar. W 1964 r. megabajt RAM-u kosztował około 400 000 dolarów. Dzisiaj każdy komputer osobisty zawiera wiele megabajtów RAM-u i przez większość czasu większość pamięci jest nieużywana. Trudno jest przewidzieć spadek cen masowo produkowanych urządzeń elektronicznych.

Z drugiej strony, jeżeli dysponujemy maszyną, która może wykonywać tylko jeden proces (lub tylko kilka procesów), musimy rozważyć możliwości po kolej. Można by sobie wyobrazić modyfikację evaluatora, polegającą na tym, że za każdym razem gdy napotykamy niedeterministyczny wybór, wybieramy losowo jedną z gałęzi, którą podążamy. Jednak losowy wybór może łatwo prowadzić do niepowodzenia. Moglibyśmy próbować w kółko uruchamiać evaluator dokonujący losowych wyborów i mieć nadzieję, że znajdzie on ścieżkę prowadzącą do wyniku, jednak lepiej jest *systematycznie przeszukiwać wszystkie możliwe ścieżki*. Evaluator *amb*, który opracujemy i którym będziemy się zajmować w niniejszym podrozdziale, będzie realizował systematyczne przeszukiwanie w następujący sposób: Gdy evaluator napotyka użycie *amb*, początkowo wybiera pierwszą możliwość. Decyzja ta może prowadzić do dalszych wyborów. Evaluator za każdym razem będzie początkowo wybierał pierwszą możliwość. Jeśli dokonane wybory prowadzą do niepowodzenia, evaluator automagicznie<sup>45</sup> *nawraca* do ostatniego wyboru i próbuje kolejną możliwość. Jeśli w którymś z niedeterministycznych wyborów zostaną wyczerpane wszystkie możliwości, evaluator wycofa się do poprzedniego wyboru i będzie kontynuował od tego miejsca. Proces ten prowadzi do strategii przeszukiwania znanej jako *przeszukiwanie w głąb* (ang. *depth-first search*) lub *chronologiczne przeszukiwanie z nawrotami* (ang. *chronological backtracking*)<sup>46</sup>.

<sup>45</sup> Automagicznie: „Automatycznie, ale w sposób, którego z pewnych względów (zwykle ponieważ jest to zbyt skomplikowane, zbyt paskudne lub zbyt trywialne) mówiący to nie ma ochoty wyjaśniać” [84, 96].

<sup>46</sup> Scalenie strategii automatycznego przeszukiwania i języków programowania ma długą i burzliwą historię. Pierwsze propozycje eleganckiego zakodowania algorytmów niedeterministycznych w języku programowania z automatycznym przeszukiwaniem z nawrotami pochodziły od Roberta Floyda [28]. Carl Hewitt [49] wymyślił język programowania o nazwie Planner, który jawnie wspomagał automatyczne chronologiczne wyszukiwanie z nawrotami za pomocą wbudowanej strategii przeszukiwania w głąb. Sussman, Winograd i Charniak [101] zaimplementowali podzbior tego języka, o nazwie MicroPlanner, który zastosowano przy wspomaganiu rozwiązywania problemów i planowania działania robotów. Podobne koncepcje wywodzące się z logiki i dowodzenia twierdzeń doprowadziły do powstania w Edynburgu i Marsylii eleganckiego języka o nazwie Prolog (który omówimy w podrozdziale 4.4). Po odpowiedniej dawce frustracji związanych z automatycznym przeszukiwaniem McDermott i Sussman [75] opracowali język o nazwie Conniver, który zawiera mechanizmy pozwalające użytkownikowi na sterowanie strategiami automatycznego wyszukiwania. Okazał się on jednak trudny w obsłudze, w związku z czym Sussman i Stallman [98], badając metody symbolicznej analizy obwodów elektronicznych, znaleźli bardziej poręczne rozwiązanie. Opracowali schemat niechronologicznego przeszukiwania z nawrotami, oparty na odnajdowaniu zależności logicznych między faktami — technikę, która była potem znana jako *przeszukiwanie z nawrotami sterowane zależściami* (ang. *dependency-directed backtracking*). Choć metoda ta była złożona, dawała programy o rozsądnej efektywności, gdyż wykonywała niewiele nadmiarowych przeszukiwań. Doyle [22] i McAllester [69, 70] uogólnili i wyjaśnili metodę Stallmana i Sussmana, rozwijając nowy paradymat formułowania przeszukiwań, który jest teraz znany jako *utrzymywanie wiarygodności* (ang. *truth maintenance*). Wszystkie nowoczesne systemy rozwiązywania problemów opierają się na jakiejś formie systemów utrzymywania wiarygodności. W książce Forbusa i deKleera [29] można znaleźć omówienie eleganckich sposobów budowania sys-

### Pętla sterująca

Pętla sterująca evaluatora `amb` ma pewne niezwykłe własności. Wczytuje wyrażenie i wypisuje wynik pierwszego obliczenia, które nie kończy się niepowodzeniem, jak to zostało pokazane wcześniej na przykładzie procedury `prime-sum-pair`. Gdybyśmy chcieli zobaczyć wartość kolejnego udanego obliczenia, możemy poprosić interpreter, aby dokonał nawrotu i spróbował wygenerować drugie obliczenie, które nie kończy się porażką. Sygnalizujemy to za pomocą symbolu `try-again`. Jeśli podamy jakiekolwiek wyrażenie poza `try-again`, interpreter rozpoczęcie rozwiązywania nowego problemu, porzucając niezbadane możliwości z poprzedniego problemu. Oto przykład interakcji:

```
;;; Amb-Eval wprowadź wyrażenie:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Rozpoczęcie rozwiązywania nowego problemu
;;; Amb-Eval wartość:
(3 20)

;;; Amb-Eval wprowadź wyrażenie:
try-again
;;; Amb-Eval wartość:
(3 110)

;;; Amb-Eval wprowadź wyrażenie:
try-again
;;; Amb-Eval wartość:
(8 35)

;;; Amb-Eval wprowadź wyrażenie:
try-again
;;; Poniższe wyrażenie nie ma więcej wartości
(prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))

;;; Amb-Eval wprowadź wyrażenie:
(prime-sum-pair '(19 27 30) '(11 36 58))
;;; Rozpoczęcie rozwiązywania nowego problemu
;;; Amb-Eval wartość:
(30 11)
```

### Ćwiczenie 4.35

Napisz procedurę `an-integer-between`, która daje w wyniku liczbę całkowitą z danego zakresu. Procedurę tę można w następujący sposób zastosować do zaimple-

mów utrzymywania wiarygodności oraz zastosowań wykorzystujących techniki utrzymywania wiarygodności. Zabih, McAllester i Chapman [110] opisują rozszerzenie języka Scheme o nie-determinizm oparte na formie `amb`; jest ono podobne do interpretera opisanego w niniejszym podręczniku, ale jest bardziej wyrafinowane, gdyż zamiast chronologicznego przeszukiwania z nawrotami używa przeszukiwania z nawrotami sterowanego zależnościami. Wprowadzenie do obydwu rodzajów przeszukiwania z nawrotami można znaleźć w książce Winstona [109].

mentowania procedury wyznaczającej trójkę pitagorejskie, tzn. takie trójkę  $(i, j, k)$  liczb całkowitych z zadanego zakresu, że  $i \leq j$  oraz  $i^2 + j^2 = k^2$ :

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k)))))
```

### Ćwiczenie 4.36

W ćwiczeniu 3.69 był omówiony sposób generowania strumienia *wszystkich* trójk pitagorejskich bez górnego ograniczenia na wartość sprawdzanych liczb. Wyjaśnij, dlaczego prosta zamiana *an-integer-between* na *an-integer-starting-from* w procedurze z ćwiczenia 4.35 nie jest odpowiednim sposobem generowania dowolnych trójk pitagorejskich. Napisz procedurę, która faktycznie tego dokona. (To znaczy napisz procedurę, dla której wielokrotne wprowadzanie *try-again* spowodowałoby wygenerowanie w końcu każdej trójkę pitagorejskiej).

### Ćwiczenie 4.37

Ben Bajerbit twierdzi, że następująca metoda generowania trójk pitagorejskich jest bardziej efektywna niż ta z ćwiczenia 4.35. Czy ma rację? (Wskazówka: Rozważ liczbę możliwości, które muszą być zbadane).

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (hsq (* high high)))
  (let ((j (an-integer-between i high)))
    (let ((ksq (+ (* i i) (* j j))))
      (require (>= hsq ksq))
      (let ((k (sqrt ksq)))
        (require (integer? k))
        (list i j k))))))
```

## 4.3.2. Przykłady programów niedeterministycznych

W punkcie 4.3.3 opiszymy implementację evaluatora *amb*. Najpierw jednak podamy kilka przykładów jego zastosowania. Korzyść wynikająca z programowania niedeterministycznego polega na tym, że możemy ukryć szczegóły realizacji wyszukiwania, wyrażając tym samym nasz program na wyższym poziomie abstrakcji.

### Łamigłówki logiczne

Następująca łamigłówka (zaczerpnięta z książki Dinesmana [20]) jest typowym przykładem szerokiej klasy prostych łamigłówek logicznych:

Baker, Cooper, Fletcher, Miller i Smith mieszkają na różnych piętrach tego samego domu, który ma pięć kondygnacji. Baker nie mieszka na

samej górze. Cooper nie mieszka na samym dole. Fletcher nie mieszka ani na samej górze, ani na samym dole. Miller mieszka wyżej niż Cooper. Smith nie mieszka ani tuż nad, ani tuż pod Fletcherem. Fletcher nie mieszka ani tuż nad, ani tuż pod Cooperem. Gdzie każdy z nich mieszka?

Możemy w prosty sposób określić, na którym piętrze mieszka każdy z nich, wyliczając wszystkie możliwości i narzucając na nie podane ograniczenia<sup>47</sup>:

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
      (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

W wyniku obliczenia wyrażenia `(multiple-dwelling)` otrzymujemy

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

Mimo że ta prosta procedura działa, jest ona bardzo wolna. W ćwiczeniach 4.39 i 4.40 omawiamy kilka możliwych ulepszeń.

---

<sup>47</sup> Poniższy program używa następującej procedury do sprawdzenia, czy wszystkie elementy listy są różne:

```
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
```

Procedura `member` jest taka jak `memq` z wyjątkiem tego, że porównuje elementy za pomocą `equal?` zamiast `eq?`.

**Ćwiczenie 4.38**

Zmodyfikuj procedurę `multiple-dwelling` tak, aby pomijała warunek, że Smith i Fletcher nie mieszkają na sąsiadujących ze sobą piętrach. Ile rozwiązań ma tak zmodyfikowana łamigłówka?

**Ćwiczenie 4.39**

Czy kolejność ograniczeń w procedurze `multiple-dwelling` ma wpływ na wynik? Czy ma ona wpływ na czas potrzebny do wyznaczenia wyniku? Jeśli uważasz, że ma wpływ, przedstaw szybszy program powstały w wyniku zmiany kolejności ograniczeń. Jeśli uważasz, że kolejność nie ma znaczenia, uzasadnij swój pogląd.

**Ćwiczenie 4.40**

Ille jest rozmieszczeń osób na piętrach w przykładowym problemie z mieszkaniem zarówno przed, jak i po narzuceniu ograniczenia, że każdy ma mieszkać na osobnym piętrze? Generowanie najpierw wszystkich możliwych rozmieszczeń osób na piętrach, a następnie eliminowanie ich za pomocą przeszukiwania z nawrotami jest bardzo nieefektywne. Na przykład większość ograniczeń dotyczy rozmieszczenia tylko jednej lub dwóch osób, mogą więc być uwzględnione przed wybraniem pięter dla wszystkich osób. Napisz i przedstaw dużo bardziej efektywną procedurę niedeterministyczną, która rozwiązuje ten problem, generując tylko te możliwości, których nie wykluczają uprzednio ograniczenia. (Wskazówka: Będzie to wymagać zagnieźdzenia wyrażeń let).

**Ćwiczenie 4.41**

Napisz zwykły program w języku Scheme rozwiązujący łamigłówkę z mieszkańcami domu.

**Ćwiczenie 4.42**

Rozwiąż następującą łamigłówkę „Klamczuchy” (pochodzącą z książki Phillipsa [81]):

Pięć uczennic wzięło udział w egzaminie. Ich rodzice — jak sądziły — okazywali nadmierne zainteresowanie wynikami. Dlatego też ustaliły, że pisząc do domu o egzaminie, każda z nich napisze jedno prawdziwe stwierdzenie i jedno fałszywe. Oto odpowiednie fragmenty ich listów:

- Betty: „Kitty była druga na egzaminie. Ja byłam zaledwie trzecią”.
- Ethel: „Miło Wam będzie usłyszeć, że byłam najlepsza. Joan była druga”.
- Joan: „Byłam trzecia, a biedna Ethel była ostatnia”.
- Kitty: „Okazało się, że byłam druga. Mary była zaledwie czwarta”.
- Mary: „Byłam czwarta. Najlepsza była Betty”.

Jaka była w rzeczywistości kolejność wyników dziewcząt?

### Ćwiczenie 4.43

Rozwiąż, za pomocą evaluatora amb następującą łamigłówkę<sup>48</sup>:

Ojciec Mary Ann Moore miał jacht, tak samo jak każdy z jego czterech przyjaciół: pułkownik Downing, pan Hall, sir Barnacle Hood i dr Parker. Każdy z pięciu panów miał jedną córkę i każdy nazwał swój jacht imieniem córki jednego z pozostałych panów. Jacht sir Barnacle'a nazywa się Gabrielle, pana Moore'a — Lorne, pana Halla — Rosalinda. Jacht Melissa, którego właścicielem jest pułkownik Downing, otrzymał imię córki sir Barnacle'a. Ojciec Gabrielli jest właścicielem jachtu nazwanego imieniem córki dr. Parkera. Kto jest ojcem Lorni?

Spróbuj napisać program, który działałby efektywnie (zob. ćwiczenie 4.40). Ustal, ile byłoby rozwiązań, gdybyśmy nie wiedzieli, że Mary Ann ma na nazwisko Moore.

### Ćwiczenie 4.44

W ćwiczeniu 2.42 opisaliśmy problem „ośmiu hetmanów” polegający na takim umieszczeniu ośmiu hetmanów na szachownicy, aby żadne dwa z nich się nie szachowały. Napisz program niedeterministyczny rozwiązujący ten problem.

### Analiza składniowa języka naturalnego

Programy zaprojektowane tak, aby przyjmowały na wejściu dane w języku naturalnym, zwykle rozpoczynają działanie od przeprowadzenia *analizy składniowej* danych wejściowych, tzn. próbując dopasować dane wejściowe do pewnej struktury gramatycznej. Możemy na przykład starać się rozpoznawać proste zdania składające się z rodzajnika (ang. *article*)<sup>\*</sup>, po którym następuje rzeczownik (ang. *noun*) oraz czasownik (ang. *verb*), takie jak „*The cat eats*” („Kot je”). Chcąc dokonać takiej analizy, musimy potrafić rozróżnić, jakimi częściami mowy są poszczególne słowa. Moglibyśmy zacząć od zdefiniowania kilku list klasyfikujących różne słowa<sup>49</sup>:

```
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
```

Potrzebujemy również *gramatyki*, tzn. zestawu reguł opisujących, w jaki sposób części gramatyczne są zbudowane z prostszych elementów. Bardzo prosta

<sup>48</sup> Pochodzi ona z broszury pt. „Problematical Recreations” opublikowanej w 1960 r. przez Litton Industries, w której jest ona przypisywana do *Kansas State Engineer*.

\* W języku polskim nie występują rodzajniki. W niniejszym podpunkcie jest mowa o dwóch rodzajnikach: „*a*” i „*the*”, którym odpowiadają zaimki „jakiś/jakaś/jakieś” oraz „ten/ta/to” (przyp. tłum.).

<sup>49</sup> Używamy tu konwencji, zgodnie z którą pierwszy element każdej listy wyznacza część mowy pozostałych słów występujących na liście.

gramatyka może określać, że każde zdanie składa się z dwóch części — wyrażenia rzeczownikowego (ang. *noun-phrase*) i czasownika (ang. *verb*) — oraz że wyrażenie rzeczownikowe składa się z rodzajnika i rzeczownika. W takiej gramatyce zdanie „The cat eats” po zanalizowaniu ma następującą strukturę:

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))
```

Możemy dokonać takiej analizy za pomocą prostego programu, w którym dla każdej reguły gramatycznej mamy osobną procedurę. Chcąc zanalizować składnię zdania, rozpoznajemy jego dwa składniki i przekazujemy jako wynik listę zawierającą znacznik **sentence** oraz te dwa składniki zdania:

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

Wyrażenie rzeczownikowe analizujemy podobnie, rozpoznając rodzajnik oraz występujący po nim rzeczownik:

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

Na najniższym poziomie analiza składniowa sprowadza się do wielokrotnego sprawdzania, czy kolejne, niewczytane jeszcze słowo występuje na liście słów stanowiących określona część mowy. Implementujemy to, zapamiętując w zmiennej globalnej **\*unparsed\*** niezanalizowane jeszcze słowa wejściowe. Za każdym razem, gdy sprawdzamy kolejne słowo, wymagamy, aby **\*unparsed\*** było niepuste i zaczynało się od słowa z określonej listy. W takim wypadku usuwamy słowo z listy **\*unparsed\*** i przekazujemy je jako wynik wraz z częścią mowy (wziętą z początku danej listy), którą ono stanowi<sup>50</sup>:

```
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))
```

---

<sup>50</sup> Zwróćmy uwagę, że **parse-word** używa **set!** do zmiany listy niezanalizowanych jeszcze słów. Aby całość działała, evaluator **amb** musi w trakcie nawracania cofać rezultaty operacji **set!**.

Do rozpoczęcia analizy składniowej wystarczy, że ustawimy `*unparsed*` jako całe wejście, spróbujemy zanalizować zdanie i sprawdzimy, czy nic nie pozostało:

```
(define *unparsed* '())

(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))
```

Możemy teraz wypróbować analizator składniowy i sprawdzić, czy zadziała dla naszego prostego próbnego zdania:

```
;;; Amb-Eval wprowadź wyrażenie:
(parse '(the cat eats))
;;; Rozpoczęcie rozwiązywania nowego problemu
;;; Amb-Eval wartość:
(sentence (noun-phrase (article the) (noun cat)) (verb eats))
```

Evaluator `amb` jest tutaj szczególnie przydatny ze względu na wygodę wyrażania warunków analizy składniowej za pomocą `require`. Automatyczne przeszukiwanie z nawrotami rzeczywiście się opłaca, zwłaszcza gdy rozważamy bardziej złożone gramatyki, w których jednostki gramatyczne mogą być rozkładane na wiele możliwości.

Dodajmy do naszej gramatyki listę przyimków (ang. *prepositions*):

```
(define prepositions '(prep for to in by with))
```

i zdefiniujmy wyrażenie przyimkowe (ang. *prepositional phrase*), na przykład „for the cat” („dla kota”), jako przyimek, po którym następuje wyrażenie rzeczownikowe:

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
        (parse-word prepositions)
        (parse-noun-phrase)))
```

Możemy teraz określić, że zdanie składa się z wyrażenia rzeczownikowego, po którym następuje wyrażenie czasownikowe, gdzie wyrażenie czasownikowe może być albo czasownikiem, albo wyrażeniem czasownikowym z dodanym wyrażeniem przyimkowym<sup>51</sup>:

---

<sup>51</sup> Zauważmy, że definicja ta jest rekurencyjna — po czasowniku może występować dowolna liczba wyrażeń przyimkowych.

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-verb-phrase)))

(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                            verb-phrase
                            (parse-prepositional-phrase))))))
  (maybe-extend (parse-word verbs)))
```

Gdy już jesteśmy przy tym, możemy rozwinać definicję wyrażenia rzeczownikowego tak, aby dopuszczała wyrażenia postaci „a cat in the class” („kot w klasie”). To, co do tej pory nazywaliśmy wyrażeniem rzeczownikowym, będziemy nazywać prostym wyrażeniem rzeczownikowym (ang. *simple noun phrase*), a wyrażenie rzeczownikowe będzie albo prostym wyrażeniem rzeczownikowym, albo wyrażeniem rzeczownikowym, po którym następuje wyrażenie przyimkowe:

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))

(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
                            noun-phrase
                            (parse-prepositional-phrase))))))
  (maybe-extend (parse-simple-noun-phrase)))
```

Nasza nowa gramatyka umożliwia analizę składniową bardziej złożonych zdań. Na przykład wynikiem

```
(parse '(the student with the cat sleeps in the class))
```

jest

```
(sentence
  (noun-phrase
    (simple-noun-phrase (article the) (noun student))
    (prep-phrase (prep with)
      (simple-noun-phrase
        (article the) (noun cat))))
```

```
(verb-phrase
  (verb sleeps)
  (prep-phrase (prep in)
    (simple-noun-phrase
      (article the) (noun class)))))
```

Zauważmy, że dla określonych danych wejściowych może istnieć więcej niż jeden poprawny rozbiór gramatyczny. W zdaniu „The professor lectures to the student with the cat” („Profesor wykłada dla studenta z kotem”) profesor może wykładać razem z kotem albo student może mieć kota. Nasz niedeterministyczny program znajduje obie możliwości:

```
(parse '(the professor lectures to the student with the cat))
```

daje w wyniku

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb-phrase
      (verb lectures)
      (prep-phrase (prep to)
        (simple-noun-phrase
          (article the) (noun student))))))
  (prep-phrase (prep with)
    (simple-noun-phrase
      (article the) (noun cat)))))
```

Prosząc evaluator o to, aby spróbował jeszcze raz, otrzymujemy

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb lectures)
    (prep-phrase (prep to)
      (noun-phrase
        (simple-noun-phrase
          (article the) (noun student)))
      (prep-phrase (prep with)
        (simple-noun-phrase
          (article the) (noun cat)))))))
```

#### Ćwiczenie 4.45

Przy podanej powyżej gramatyce następujące zdanie może być rozebrane na pięć różnych sposobów: „The professor lectures to the student in the class with the cat” („Profesor wykłada dla studenta w klasie z kotem”). Podaj tych pięć rozbiórów i wyjaśnij niuanse znaczeniowe, którymi się różnią.

**Ćwiczenie 4.46**

Ewaluator z podrozdziałów 4.1 i 4.2 nie określają kolejności obliczania argumentów. Jak zobaczymy, ewaluator amb oblicza je od lewej do prawej. Wyjaśnij, dlaczego program analizy składniowej nie działałby, gdyby argumenty były obliczane w innej kolejności.

**Ćwiczenie 4.47**

Ludwik Myślicielak zaproponował, że skoro wyrażenie czasownikowe składa się albo z czasownika, albo z wyrażenia czasownikowego, po którym następuje wyrażenie przyimkowe, prościej byłoby zaimplementować procedurę `parse-verb-phrase` w następujący sposób (oraz podobnie dla wyrażeń rzeczownikowych):

```
(define (parse-verb-phrase)
  (amb (parse-word verbs)
        (list 'verb-phrase
              (parse-verb-phrase)
              (parse-prepositional-phrase))))
```

Czy to zadziała? Czy działanie programu uległoby zmianie, gdybyśmy zmienili kolejność wyrażeń składowych amb?

**Ćwiczenie 4.48**

Rozszerz podaną powyżej gramatykę, aby opisywała bardziej złożone zdania. Móglbyś na przykład rozszerzyć wyrażenia rzeczownikowe i czasownikowe o przymiotniki i przysłówki lub też mógłbyś opisać zdania złożone<sup>52</sup>.

**Ćwiczenie 4.49**

Liz P. Haker bardziej interesuje się generowaniem ciekawych zdań niż ich analizą składniową. Sądzi ona, że zmieniając w prosty sposób procedurę `parse-word`, tak aby ignorowała „zdanie wejściowe” oraz zawsze kończyła się powodzeniem i generowała odpowiednie słowo, możemy użyć naszego programu do generowania zdań zamiast do ich analizy. Zaimplementuj rozwiązańe Liz i podaj kilka pierwszych zdań, jakie można wygenerować<sup>53</sup>.

<sup>52</sup> Tego rodzaju gramatyka może być dowolnie skomplikowana, a mimo to będzie jedynie prostym modelem w porównaniu z bogactwem rzeczywistego języka. Zrozumienie przez komputer rzeczywistego języka naturalnego wymaga misternego połączenia analizy składniowej i interpretacji znaczenia. Jednakże nawet proste analizatory składniowe mogą być przydatne, pomagając w tworzeniu elastycznych języków poleceń dla takich programów jak systemy wyszukiwania informacji. Winston [109] omawia różne obliczeniowe sposoby podejścia do rozumienia rzeczywistego języka, a także zastosowanie prostych gramatyk w językach poleceń.

<sup>53</sup> Choć rozwiązańe Liz działa poprawnie (i jest zaskakująco proste), zdania jakie generuje są trochę nudne — nie stanowią one zbyt interesującej próbki rozważanego języka. W rzeczywistości gramatyka języka jest w wielu miejscach wysoce rekurencyjna i technika Liz „wpada” w jedną z takich rekurencji i utyka. Zobacz ćwiczenie 4.50, żeby dowiedzieć się, jak sobie radzić z takim problemem.

### 4.3.3. Implementacja ewaluatora amb

Obliczenie wyrażenia w zwykłym języku Scheme może dać w wyniku wartość, może się nigdy nie zakończyć lub może spowodować wystąpienie błędu. W niedeterministycznym języku Scheme obliczenie wyrażenia może dodatkowo utknąć w ślepym zaułku — wówczas obliczenie musi zostać wycofane do poprzedniego niedeterministycznego wyboru. Ten dodatkowy przypadek komplikuje interpretację niedeterministycznego języka Scheme.

Skonstruujemy ewaluator `amb` niedeterministycznego języka Scheme, modyfikując ewaluator z wydzieloną analizą składniową z punktu 4.1.7<sup>54</sup>. Tak jak w ewaluatorze z wyodrębnioną analizą składniową, obliczenie wyrażenia dokonuje się poprzez wywołanie procedury wykonawczej, która powstaje jako wynik analizy danego wyrażenia. Różnica między interpretacją zwykłego i niedeterministycznego języka Scheme będzie się całkowicie zawierać w procedurach wykonawczych.

#### Procedury wykonawcze i kontynuacje

Przypomnijmy, że procedury wykonawcze w zwykłym ewaluatorze mają tylko jeden argument: środowisko wykonania. Natomiast procedury wykonawcze ewaluatora `amb` będą miały trzy argumenty: środowisko oraz dwie procedury nazywane *procedurami kontynuacji* (lub krócej *kontynuacjami*; ang. *continuations*). Obliczenie wyrażenia będzie się kończyć wywołaniem jednej z kontynuacji — jeśli wynikiem obliczenia będzie wartość, to wywoływana będzie dla tej wartości *kontynuacja udanego obliczenia* (ang. *success continuation*), a jeśli obliczenie utknie w ślepym zaułku, to wywoływana będzie *kontynuacja niepowodzenia* (ang. *failure continuation*). Konstruowanie i wywoływanie odpowiednich kontynuacji stanowi mechanizm, za pomocą którego niedeterministyczny ewaluator implementuje przeszukiwanie z nawrotami.

Do zadań procedury kontynuacji udanego obliczenia należy odebranie wartości i kontynuowanie obliczenia. Razem z tą wartością kontynuacja udanego obliczenia otrzymuje kontynuację niepowodzenia, którą powinna później wywołać, jeśli użycie otrzymanej wartości prowadzi do ślepego zaułka.

Do zadań procedury kontynuacji niepowodzenia należy wypróbowanie kolejnej gałęzi procesu niedeterministycznego. Istota języka niedeterministycznego polega na tym, że wyrażenia mogą reprezentować wybory spośród wielu możliwości. Obliczenie takiego wyrażenia musi przebiegać zgodnie z jedną ze wskazanych możliwości, mimo że nie wiadomo z góry, które możliwości prowadzą do akceptowalnych wyników. Aby sobie z tym poradzić, ewaluator wybiera jed-

<sup>54</sup> W podrozdziale 4.2 zdecydowaliśmy się zaimplementować leniwy ewaluator jako modyfikację zwykłego ewaluatora metacyjkicznego z punktu 4.1.1. Z kolei podstawą do opracowania ewaluatora `amb` będzie ewaluator z wydzieloną analizą składniową z punktu 4.1.7, gdyż procedury wykonawcze tego ewaluatora stanowią wygodny szkielet do zaimplementowania przeszukiwania z nawrotami.

ną z możliwości i przekazuje jej wartość do kontynuacji udanego obliczenia. Razem z tą wartością evaluator konstruuje i przekazuje kontynuację niepowodzenia, która może być później wywołana w celu wybrania innej możliwości.

Niepowodzenie jest wykrywane w trakcie obliczeń (tzn. wywoływana jest kontynuacja niepowodzenia), gdy program użytkownika jawnie odrzuci aktualną linię ataku (np. wywołanie `require` może spowodować wykonanie `(amb)` — wyrażenia, którego obliczenie zawsze kończy się porażką — zob. punkt 4.3.1). Kontynuacja niepowodzenia, którą w takim momencie dysponujemy, spowoduje, że ostatni wybór niedeterministyczny spróbuje inną możliwość. Jeśli nie ma już innych możliwości do rozpatrzenia, ma miejsce niepowodzenie związane z jeszcze wcześniejszym wyborem niedeterministycznym itd. Kontynuacje niepowodzenia są również wywoływane przez pętlę sterującą w odpowiedzi na polecenie `try-again` wydane w celu znalezienia innej wartości wyrażenia.

Dodatkowo, jeśli na gałęzi obliczeń wychodzącej z wyboru niedeterministycznego pojawi się operacja powodująca efekty uboczne (taka jak przypisanie do zmiennej), a proces znajdzie się w ślepym zaułku, to przed dokonaniem nowego wyboru może być konieczne cofnięcie efektów ubocznych. Realizowane jest to w ten sposób, że operacje powodujące efekty uboczne tworzą kontynuacje niepowodzeń, które cofają te efekty uboczne i rozprzestrzeniają niepowodzenie.

Podsumowując, kontynuacje niepowodzeń są tworzone przez

- wyrażenia `amb` — w celu dostarczenia mechanizmu zapewniającego wybranie innych możliwości, jeśli bieżący wybór dokonany przez wyrażenie `amb` prowadzi do ślepego zaułka;
- pętlę sterującą — w celu dostarczenia mechanizmu sygnalizującego niepowodzenie, gdy wszystkie możliwości wyboru zostaną wyczerpane;
- przypisania — w celu przechwytywania niepowodzeń i cofania efektów przypisań w trakcie nawracania przeszukiwania.

Niepowodzenia powstają tylko wtedy, gdy obliczenie znajduje się w ślepym zaułku. Zachodzi to, gdy

- program użytkownika wykona `(amb)`;
- użytkownik zleci pętli sterującej wykonanie polecenia `try-again`.

Kontynuacje niepowodzeń są również wywoływane w trakcie przetwarzania niepowodzeń:

- Gdy kontynuacja niepowodzenia utworzona przez przypisanie cofnie efekty uboczne, wówczas wywołuje ona kontynuację niepowodzenia, które przechwy-

ciła, w celu rozprzestrzenienia niepowodzenia aż do niedeterministycznego wyboru, który prowadził do tego przypisania, lub aż do pętli sterującej.

- Gdy kontynuacja niepowodzenia amb wyczerpie wszystkie możliwości, wówczas wywołuje ona kontynuację niepowodzenia (która była pierwotnie przekazana amb) w celu rozprzestrzenienia niepowodzenia aż do poprzedniego niedeterministycznego wyboru lub aż do pętli sterującej.

## Struktura ewaluatora

Procedury reprezentacji składni i danych ewaluatora amb, a także podstawa-wa procedura analyze, są identyczne jak te w ewaluatorze z punktu 4.1.7, z wyjątkiem tego, że potrzebujemy dodatkowych procedur składniowych do rozpoznawania formy specjalnej amb<sup>55</sup>:

```
(define (amb? exp) (tagged-list? exp 'amb))

(define (amb-choices exp) (cdr exp))
```

Musimy również dodać klauzulę rozpoznającą tę formę specjalną w analizie przypadków w procedurze analyze:

```
((amb? exp) (analyze-amb exp))
```

Główna procedura ambeval (podobna do wersji eval przedstawionej w punkcie 4.1.7) analizuje dane wyrażenie i stosuje utworzoną procedurę wy-konawczą do zadanego środowiska oraz dwóch danych kontynuacji:

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

Kontynuacja udanego obliczenia to procedura o dwóch argumentach: wła-snie uzyskanej wartości i kontynuacji niepowodzenia, której należy użyć, jeśli uzyskana wartość prowadzi do późniejszego niepowodzenia. Kontynuacja nie-powodzenia to procedura bezargumentowa. Tak więc ogólna postać procedury wy-konawczej jest następująca:

```
(lambda (env succeed fail)
  ;;= succeed jest postaci (lambda (value fail) ...)
  ;;= fail jest postaci (lambda () ...)
  ...)
```

---

<sup>55</sup> Zakładamy, że ewaluator umożliwia użycie formy let (zob. ćwiczenie 4.22), której używa-liśmy w przedstawionych programach niedeterministycznych.

Na przykład wywołanie

```
(ambeval <wyrażenie>
  the-global-environment
  (lambda (value fail) value)
  (lambda () 'failed))
```

będzie powodowało próbę obliczenia wartości danego wyrażenia i będzie dawało w wyniku albo wartość całego wyrażenia (jeśli obliczenie się uda), albo symbol failed (jeśli obliczenie kończy się porażką). Pokazane dalej wywołanie ambeval w pętli sterującej korzysta z dużo bardziej skomplikowanych procedur kontynuacji, które kontynuują działanie pętli i obsługują polecenia try-again.

Złożoność evaluatora amb wynika w znacznym stopniu z mechanizmu przekazywania kontynuacji między wzajemnymi wywołaniami procedur wykonawczych. Przeglądając poniższy kod, powinieneś porównać każdą procedurę wykonawczą z odpowiadającą jej procedurą zwykłego evaluatora z punktu 4.1.7.

### Wyrażenia proste

Procedury wykonawcze dla najprostszych rodzajów wyrażeń są zasadniczo takie same jak w zwykłym evaluatorze, z wyjątkiem konieczności obsługi kontynuacji. Procedury te po prostu kończą pomyślnie działanie z wartością wyrażenia, przekazując dalej otrzymaną kontynuację niepowodzenia.

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env succeed fail)
      (succeed qval fail)))

(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env)
             fail)))

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env)
               fail))))
```

Zauważmy, że sprawdzanie wartości zmiennej zawsze się „udaje”. Jeśli procedura lookup-variable-value nie może znaleźć zmiennej, to jak zwykle

sygnalizuje błąd. Jednak takie „niepowodzenie” wskazuje na błąd w programie — odwołanie do zmiennej wolnej; nie oznacza ono, że zamiast aktualnie sprawdzanego niedeterministycznego wyboru powinniśmy spróbować innego.

### Wyrażenia warunkowe i ciągi wyrażeń

Wyrażenia warunkowe są również obsługiwane podobnie jak w zwykłym evaluatorze. Procedura wykonawcza generowana przez `analyze-if` wywołuje procedurę wykonawczą predykatu `pproc`, przekazując jej kontynuację udanego obliczenia, która sprawdza, czy predykat ma wartość prawda, i stosownie do tego wykonuje dalej albo następnik, albo alternatywę. Jeśli wykonanie `pproc` kończy się niepowodzeniem, wywoływana jest pierwotna kontynuacja niepowodzenia wyrażenia `if`.

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
        ;; kontynuacja udanego obliczenia
        ;; wartości predykatu pred-value
        (lambda (pred-value fail2)
          (if (true? pred-value)
              (cproc env succeed fail2)
              (aproc env succeed fail2)))
        ;; kontynuacja obliczenia wartości predykatu
        ;; kończącego się niepowodzeniem
        fail))))
```

Ciągi wyrażeń są także obsługiwane w taki sam sposób jak w poprzednim evaluatorze, z wyjątkiem machinacji w podprocedurze `sequentially` potrzebnych do przekazywania kontynuacji. Mianowicie, chcąc sekwencyjnie wykonać najpierw a, a potem b, wywołujemy a, przekazując kontynuację udanego obliczenia, która wywoła b.

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
        ;; kontynuacja udanego wywołania a
        (lambda (a-value fail2)
          (b env succeed fail2)))
        ;; kontynuacja wywołania a zakończonego niepowodzeniem
        fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
```

```

first-proc
  (loop (sequentially first-proc (car rest-procs))
        (cdr rest-procs)))
(let ((procs (map analyze exps)))
  (if (null? procs)
      (error "Ciąg pusty -- ANALYZE"))
  (loop (car procs) (cdr procs))))

```

### Definicje i przypisania

Definicje stanowią kolejny przypadek, kiedy musimy zadać sobie trochę trudu, zajmując się kontynuacjami, ponieważ musimy najpierw obliczyć wyrażenie określające definiowaną wartość, a dopiero potem możemy faktycznie zdefiniować nową zmienną. Aby tego dokonać, wywołujemy procedurę wykonawczą vproc obliczającą definiowaną wartość, przekazując jej środowisko, kontynuację udanego obliczenia oraz kontynuację niepowodzenia. Jeśli wykonanie vproc kończy się pomyślnie, uzyskujemy wartość val definiowanej zmiennej, zmienna jest definiowana i pomyślne zakończenie obliczenia jest przekazywane dalej:

```

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env succeed fail)
      (vproc env
            (lambda (val fail2)
              (define-variable! var val env)
              (succeed 'ok fail2))
            fail))))

```

Przypisania są ciekawsze. Jest to pierwsze miejsce, gdzie faktycznie używamy kontynuacji, a nie tylko przekazujemy je dalej. Procedura wykonawcza przypisania zaczyna działanie tak jak procedura wykonawcza definicji. Najpierw próbuje uzyskać nową wartość (obliczając vproc), która ma być przypisana do zmiennej. Jeśli obliczenie to się nie uda, całe przypisanie kończy się niepowodzeniem.

Jeśli jednak obliczenie vproc kończy się pomyślnie i przystępujemy dalej do wykonania przypisania, musimy wziąć pod uwagę możliwość, że dana gałąź obliczeń może dalej nie powieść się, co będzie wymagało od nas wycofania się z przypisania. Musimy więc zadbać o to, aby cofanie efektów przypisania było częścią nawracania w procesie przeszukiwania<sup>56</sup>.

Dokonujemy tego, przekazując vproc kontynuację udanego obliczenia (zaznaczoną poniżej komentarzem „\*1\*”), która zapamiętuje starą wartość zmien-

---

<sup>56</sup> Nie przejmowaliśmy się cofaniem definicji, ponieważ możemy założyć, że wyłuskano definicje wewnętrzne (punkt 4.1.6).

nej przed przypisaniem jej nowej wartości i kontynuowaniem obliczeń. Kontynuacja niepowodzenia, która jest przekazywana razem z wynikiem przypisania (zaznaczona poniżej komentarzem „\*2\*”) przywraca poprzednią wartość zmiennej przed kontynuowaniem niepowodzenia. Inaczej mówiąc, udane przypisanie udostępnia kontynuację niepowodzenia, która przechwytuje ewentualne późniejsze niepowodzenie — zamiast wywołania `fail2` wywoływana jest ta kontynuacja, cofa ona efekty przypisania, po czym faktycznie wywołuje `fail2`.

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
        (lambda (val fail2) ; *1*
          (let ((old-value
                 (lookup-variable-value var env)))
            (set-variable-value! var val env)
            (succeed 'ok
              (lambda () ; *2*
                (set-variable-value! var
                  old-value
                  env)
                (fail2))))
          fail))))
```

### Zastosowania procedur

Procedura wykonawcza dla zastosowań nie zawiera nic nowego oprócz złożoności związanej z techniką obsługi kontynuacji. Złożoność ta pojawia się w `analyze-application` ze względu na konieczność śledzenia kontynuacji udanych obliczeń i niepowodzeń w miarę obliczania kolejnych argumentów. Do obliczenia listy argumentów używamy procedury `get-args` zamiast prostej procedury `map`, jak miało to miejsce w zwykłym evaluatorze.

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
        (lambda (proc fail2)
          (get-args aprocs
            env
            (lambda (args fail3)
              (execute-application
                proc args succeed fail3)
              fail2))
          fail))))
```

Zwróćmy uwagę, jak w procedurze `get-args` cdrowanie wzdłuż listy `aprocs` procedur wykonawczych oraz konstruowanie listy wynikowej `args` są realizowane przez wywoływanie każdej procedury z listy `aprocs` z kontynuacją udanego obliczenia, która rekurencyjnie wywołuje `get-args`. Każde z tych wywołań rekurencyjnych `get-args` ma kontynuację udanego obliczenia, która otrzymuje wartość właśnie obliczonego wyrażenia i wartość ta jest dołączana na początku listy kumulowanych argumentów:

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
       ;; kontynuacja udanego obliczenia
       ;; tego elementu aprocs
       (lambda (arg fail2)
         (get-args (cdr aprocs)
                   env
                   ;; kontynuacja udanego obliczenia
                   ;; rekurencyjnego wywołania get-args
                   (lambda (args fail3)
                     (succeed (cons arg args)
                               fail3)))
                   fail2))
       fail)))

```

Faktyczne zastosowanie procedury, które jest wykonywane przez procedurę `execute-application`, odbywa się w ten sam sposób co w zwykłym evaluatorze z wyjątkiem konieczności obsługi kontynuacji.

```
(define (execute-application proc args succeed fail)
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                               args
                               (procedure-environment proc)))
         succeed
         fail))
        (else
         (error
          "Nieznany rodzaj procedury -- EXECUTE-APPLICATION"
          proc))))

```

## Obliczanie wyrażeń amb

Forma specjalna `amb` stanowi kluczowy element języka niedeterministycznego. Widać tutaj istotę procesu interpretacji oraz powód śledzenia kontynuacji. Procedura wykonawcza dla `amb` definiuje pętlę `try-next`, która przegląda procedury wykonawcze dla wszystkich możliwych wartości wyrażenia `amb`. Każda procedura wykonawcza jest wywoływana z kontynuacją niepowodzenia, która wypróbuje kolejną możliwość. Gdy nie ma już więcej możliwości do wypróbowania, całe wyrażenie `amb` kończy się niepowodzeniem.

```
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
             succeed
             (lambda ()
               (try-next (cdr choices)))))))
    (try-next cprocs))))
```

## Pętla sterująca

Ze względu na mechanizm pozwalający użytkownikowi na ponawianie prób obliczania wyrażenia pętla sterująca (`driver-loop`) evaluatorka `amb` jest złożona. W pętli tej jest wykorzystywana procedura o nazwie `internal-loop`, której argumentem jest procedura `try-again`. Pomysł polega na tym, aby wywołanie `try-again` powodowało przejście do kolejnej niewypróbowanej możliwości obliczenia niedeterministycznego. `Internal-loop` albo wywołuje `try-again` w odpowiedzi na polecenie `try-again` wprowadzone przez użytkownika w pętli sterującej, albo rozpoczyna nowe obliczenie, wywołując `ambeval`.

Kontynuacja niepowodzenia dla takiego wywołania `ambeval` informuje użytkownika, że nie ma więcej możliwych wartości, i ponownie wywołuje pętlę sterującą.

Kontynuacja udanego obliczenia dla wywołania `ambeval` jest bardziej subtelna. Wypisujemy uzyskaną wartość, a następnie ponownie wywołujemy pętlę wewnętrzną, przekazując jako parametr `try-again` procedurę, która będzie mogła spróbować wyznaczyć kolejną możliwość. Procedura ta jest drugim argumentem (`next-alternative`) kontynuacji udanego obliczenia. Zwykle myślimy o tym drugim argumentem jako o kontynuacji niepowodzenia, której należy użyć, jeśli bieżąca gałąź obliczeń później się nie powiedzie. W tym przypadku jednak obliczenie zakończyło się pomyślnie, możemy więc wywo-

łać inną możliwą gałąź „niepowodzenia” w celu znalezienia kolejnego udanego obliczenia.

```
(define input-prompt ";;; Amb-Eval wprowadź wyrażenie:")
(define output-prompt ";;; Amb-Eval wartość:")

(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline)
            (display ";;; Nowy problem ")
            (ambeval input
                      the-global-environment
                      ;; ambeval udane obliczenie
                      (lambda (val next-alternative)
                        (announce-output output-prompt)
                        (user-print val)
                        (internal-loop next-alternative)))
            ;; ambeval niepowodzenie
            (lambda ()
              (announce-output
                ";;; Wyrażenie nie ma więcej wartości")
              (user-print input)
              (driver-loop))))))
  (internal-loop
    (lambda ()
      (newline)
      (display ";;; Brak bieżącego problemu")
      (driver-loop))))
```

Początkowe wywołanie internal-loop korzysta z procedury try-again, która wypisuje komunikat, że nie ma bieżącego problemu do rozwiązania, i ponownie uruchamia pętlę sterującą. Takie zachowanie może wystąpić, jeśli użytkownik wprowadzi try-again, gdy nie ma żadnych obliczeń w toku.

### Ćwiczenie 4.50

Zaimplementuj nową formę specjalną ramb, która działa tak jak amb z wyjątkiem tego, że przeszukuje możliwości w losowej kolejności, a nie od lewej do prawej. Pokaż, jak może to pomóc w rozwiązyaniu problemu Liz z ćwiczenia 4.49.

### Ćwiczenie 4.51

Zaimplementuj nowy rodzaj przypisania o nazwie `permanent-set!`, którego rezultaty nie są cofane w przypadku niepowodzenia obliczeń. Możemy na przykład w następujący sposób wybierać dwa różne elementy z listy i zliczać liczbę prób potrzebnych do dokonania pomyślnego wyboru:

```
(define count 0)

(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b c))))
  (permanent-set! count (+ count 1))
  (require (not (eq? x y)))
  (list x y count))
;; Rozpoczęcie rozwiązywania nowego problemu
;; Amb-Eval wartość:
(a b 2)

;; Amb-Eval wprowadź wyrażenie:
try-again
;; Amb-Eval wartość:
(a c 3)
```

Jakie wartości byłyby wypisane, gdybyśmy użyli tutaj `set!` zamiast `permanent-set!`?

### Ćwiczenie 4.52

Zaimplementuj nową konstrukcję o nazwie `if-fail`, która pozwala użytkownikowi wychwycić niepowodzenie obliczenia wyrażenia. Argumentami `if-fail` są dwa wyrażenia. Pierwsze z tych wyrażeń jest obliczane w zwykły sposób i jeżeli obliczenie kończy się pomyślnie, to tym samym kończy się obliczenie całego wyrażenia. Jeżeli natomiast obliczenie zakończy się niepowodzeniem, to wynikiem jest wartość drugiego wyrażenia, jak w następującym przykładzie:

```
;; Amb-Eval wprowadź wyrażenie:
(if-fail (let ((x (an-element-of '(1 3 5))))
           (require (even? x))
           x)
          'all-odd)
;; Rozpoczęcie rozwiązywania nowego problemu
;; Amb-Eval wartość:
all-odd

;; Amb-Eval wprowadź wyrażenie:
(if-fail (let ((x (an-element-of '(1 3 5 8))))
           (require (even? x))
           x)
          'all-odd)
;; Rozpoczęcie rozwiązywania nowego problemu
;; Amb-Eval wartość:
```

### Ćwiczenie 4.53

Jeśli zastosuje się permanent-set! opisany w ćwiczeniu 4.51 oraz if-fail z ćwiczenia 4.52, to jaki będzie wynik obliczenia

```
(let ((pairs '()))
  (if-fail (let ((p (prime-sum-pair '(1 3 5 8) '(20 35 110)))
                (permanent-set! pairs (cons p pairs))
                (amb))
            pairs))
```

### Ćwiczenie 4.54

Gdybyśmy nie zdali sobie sprawy z tego, że require można zaimplementować za pomocą amb jako zwykłą procedurę, którą może zdefiniować użytkownik jako część programu niedeterministycznego, to musielibyśmy zaimplementować ją jako formę specjalną. Wymagałoby to zdefiniowania procedur składniowych:

```
(define (require? exp) (tagged-list? exp 'require))
(define (require-predicate exp) (cadr exp))
```

oraz nowej klauzuli rozpoznającej tę formę w procedurze analyze:

```
((require? exp) (analyze-require exp))
```

jak również procedury analyze-require obsługującej wyrażenia require. Uzupełnij następującą definicję analyze-require:

```
(define (analyze-require exp)
  (let ((pproc (analyze (require-predicate exp))))
    (lambda (env succeed fail)
      (pproc env
            (lambda (pred-value fail2)
              (if (??)
                  (??)
                  (succeed 'ok fail2)))
            fail))))
```

## 4.4. Programowanie w logice

W rozdziale 1 podkreślaliśmy, że informatyka zajmuje się wiedzą imperatywną (jak to zrobić), podczas gdy matematyka zajmuje się wiedzą deklaratywną (co to jest). Faktycznie, języki programowania wymagają, aby programista wyrażał wiedzę w postaci, która wskazuje krok po kroku metodę rozwiązywania określonego problemu. Jednakże języki wysokiego poziomu w ramach swojej implementacji zawierają pokaźną porcję wiedzy metodologicznej, która uwalnia użytkownika od zajmowania się licznymi szczegółami dotyczącymi przebiegu opisywanego obliczenia.

Większość języków programowania, w tym Lisp, koncentruje się wokół obliczania wartości funkcji matematycznych. Języki zorientowane na wyrażenia (takie jak Lisp, Fortran czy Algol) bazują na dwuznaczności wyrażeń, które nie tylko opisują wartości funkcji, ale mogą być także interpretowane jako sposoby obliczania tych wartości. Ze względu na to większość języków programowania jest silnie nastawiona na obliczenia jednokierunkowe (tzn. obliczenia o dobrze określonych danych i wynikach). Istnieją jednak zupełnie inne języki programowania, w których to nastawienie jest złagodzone. Widzieliśmy przykład takiego języka w punkcie 3.3.5, gdzie obiektami obliczeniowymi były więzy arytmetyczne. W systemie więzów kierunek i kolejność obliczeń nie są tak jednoznacznie określone; dlatego też w trakcie wykonywania obliczeń system musi bardziej szczegółowo określić, „jak” mają one przebiegać, niż miałoby to miejsce w przypadku zwykłych wyrażeń arytmetycznych. Nie oznacza to jednak, że użytkownik jest całkiem zwolniony z obowiązku dostarczenia wiedzy imperatywnej. Istnieje wiele sieci więzów, które implementują te same związki, i to właśnie do użytkownika należy wybranie spośród matematycznie równoważnych sieci takiej, która jest odpowiednia do opisu określonego obliczenia.

Ewaluator programów niedeterministycznych z podrozdziału 4.3 również oddala nas od poglądu, że programowanie polega na konstruowaniu algorytmów obliczających jednokierunkowe funkcje. W języku niedeterministycznym wyrażenia mogą mieć więcej niż jedną wartość, w wyniku czego obliczenie dotyczy relacji zamiast funkcji jednowartościowych. Programowanie w logice idzie jeszcze dalej w tym kierunku, łącząc relacyjne spojrzenie na programowanie z potężną metodą symbolicznego dopasowywania wzorców nazywaną *unifikacją* (ang. *unification*)<sup>57</sup>.

<sup>57</sup> Programowanie w logice powstało w wyniku długich badań nad automatycznym dowodzeniem twierdzeń. Pierwsze programy dowodzące potrafiły bardzo niewiele, gdyż wyczerpująco przeszukiwały przestrzeń możliwych dowodów. Wynalezienie we wczesnych latach sześćdziesiątych XX w. *algorytmu unifikacji* oraz sformułowanie *zasady rezolucji* [89] spowodowały przełom, dzięki któremu takie przeszukiwanie znalazło zastosowania. Rezolucja była na przykład zastosowana przez Greena i Raphaela [37] (zob. też [36]) jako podstawa systemu dedukcyjnego odpowiadającego na pytania. Przez większą część tego okresu badacze skupiali się na algorytmach, które znajdowały dowód, jeśli tylko jakiś istniał. Trudno było sterować takimi algorytmami i naprowadzać je na dowód. Hewitt [49] dostrzegł możliwość scalenia struktury sterującej języka programowania z operacjami systemu logicznego, co doprowadziło do powstania pracy dotyczącej metod automatycznego przeszukiwania, o której wspominamy w punkcie 4.3.1 (przypis 46). W tym samym czasie, w Marsylii, Colmerauer pracował nad systemem reguł służącym do przetwarzania języka naturalnego (zob. [15]). Opracował on język programowania Prolog służący do reprezentowania takich reguł. Kowalski [62, 63] w Edynburgu zauważył, że wykonanie programu w Prologu może być interpretowane jako dowodzenie twierdzeń (za pomocą techniki dowodowej nazywanej liniową rezolucją klauzul hornowskich). Scalenie tych dwóch podejść zaowocowało powstaniem programowania w logice. W ten sposób, jeśli chodzi o przypisywanie sobie załug w opracowaniu programowania w logice, Francuzi mogą wskazywać na powstanie Prologu na Université d'Aix Marseille, podczas gdy Brytyjczycy mogą

Podejście to, gdy zadziała, może być bardzo silną techniką pisania programów. Część tej siły pochodzi stąd, że pojedynczy fakt mówiący „co to jest” może posłużyć do rozwiązywania wielu różnych problemów zawierających różne składowe „jak to zrobić”. Jako przykład rozważmy operację `append`, której wynikiem jest lista powstała przez połączenie dwóch list będących jej argumentami. W języku proceduralnym, takim jak Lisp, moglibyśmy zdefiniować `append` za pomocą podstawowego konstruktora list `cons`, jak to uczyniliśmy w punkcie 2.2.1:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

Procedurę tę możemy traktować jako tłumaczenie na Lisp następujących dwóch reguł, z których pierwsza opisuje przypadek, gdy pierwsza lista jest pusta, a druga — gdy jest ona niepusta, czyli stanowi `cons` dwóch części:

- Dla dowolnej listy `y`: `append` listy pustej i `y` jest równe `y`.
- Dla dowolnych `u`, `v`, `y` i `z`: jeśli `append` list `v` i `y` jest równe `z`, to `append` list `(cons u v)` i `y` jest równe `(cons u z)`<sup>58</sup>.

Za pomocą procedury `append` możemy odpowiadać na takie pytania, jak:

Wyznacz `append` list `(a b)` i `(c d)`.

Jednak te same dwie reguły wystarczają również do udzielenia odpowiedzi na pytania następującego rodzaju, na które powyższa procedura nie umie odpowiedzieć:

Wyznacz taką listę `y`, że `append` `(a b)` i tej listy jest równe `(a b c d)`.

Wyznacz wszystkie takie `x` i `y`, których `append` jest równe `(a b c d)`.

Pisząc w języku programowania w logice, programista zapisuje „procedurę” `append`, podając dwie powyższe reguły dotyczące `append`. Wiedza „jak to zrobić” jest automatycznie generowana przez interpreter, dzięki czemu jedna

---

podkreślać prace powstałe na University of Edinburgh. Według ludzi z MIT programowanie w logice zostało opracowane przez te dwie grupy w ramach prób zrozumienia, o czym Hewitt pisał w swojej olśniewającej acz nieprzystępnej rozprawie doktorskiej. O historii programowania w logice można przeczytać w [90].

<sup>58</sup> Żeby zobaczyć związek między regułami i procedurami, niech `x` w procedurze (gdzie `x` jest niepustą listą) odpowiada `(cons u v)` w regule. Wówczas `z` w regule odpowiada wynikowi `append` list `(cdr x)` i `y`.

para reguł może odpowiedzieć na wszystkie trzy rodzaje pytań dotyczących append<sup>59</sup>.

Współczesne języki programowania w logice (wliczając w to język, który implementujemy tutaj) mają pokaźne braki polegające na tym, że ich ogólne metody wnioskowania „jak to zrobić” mogą prowadzić do błędnych nieskończonych pętli lub innego nieporządanego zachowania. Programowanie w logice stanowi aktywną dziedzinę badań w informatyce<sup>60</sup>.

Wcześniej w tym rozdziale badaliśmy technikę implementacji interpreterów i opisaliśmy najważniejsze elementy interpretera języka podobnego do Lispu (a właściwie interpretera dowolnego konwencjonalnego języka). Teraz sprawdzimy te koncepcje, omawiając budowę interpretera języka programowania w logice. Będziemy nazywać ten język *językiem zapytań* (ang. *query language*), gdyż jest on bardzo użyteczny do wyszukiwania informacji w bazach danych przez formułowanie *zapytań* (ang. *queries*), czyli pytań wyrażonych w tym języku. Choć język zapytań różni się bardzo od Lispu, wygodnie nam będzie opisywać go w tych samych ogólnych ramach, których używaliśmy dotychczas: jako zestaw elementów pierwotnych wraz ze środkami łączenia, które pozwalają na łączenie elementów prostych i tworzenie elementów bardziej złożonych, i środkami abstrakcji, które umożliwiają traktowanie elementów złożonych jako pojęciowych całości. Interpreter języka programowania w logice jest znacznie bardziej złożony niż interpreter takiego języka jak Lisp. Niemniej jednak zobaczymy, że nasz interpreter języka zapytań zawiera wiele takich samych elementów jak te występujące w interpreterze z podrozdziału 4.1. W szczególności będzie część „eval”, która klasyfikuje wyrażenia ze względu na ich rodzaj, i część „apply”, która implementuje mechanizm abstrakcji języka (w przypadku Lispu — procedury,

<sup>59</sup> Oczywiście nie zwalnia to całkowicie użytkownika z konieczności rozwiązywania problemu obliczania wyniku. Istnieje wiele różnych, matematycznie równoważnych zestawów reguł definiujących relację append, jednak tylko niektóre z nich mogą zamienić się w efektywne narzędzia obliczające zależności w dowolnym kierunku. Dodatkowo, czasami informacja „co” należy obliczyć nie daje żadnych wskazówek co do tego, „jak” to obliczyć. Rozważmy na przykład problem polegający na obliczeniu takiego  $y$ , że  $y^2 = x$ .

<sup>60</sup> Zainteresowanie programowaniem w logice osiągnęło szczyt we wczesnych latach osiemdziesiątych XX w., gdy rząd Japonii rozpoczął ambitny projekt zmierzający do zbudowania superszybkich komputerów zoptymalizowanych pod kątem wykonywania języków programowania w logice. Szybkość takich komputerów była mierzona w LIPS-ach (wnioskowaniach logicznych na sekundę; ang. *logical inferences per second*), a nie jak zwykle we FLOPS-ach (operacjach zmiennopozycyjnych na sekundę; ang. *floating-point operations per second*). Choć projekt zaowocował opracowaniem sprzętu i oprogramowania zgodnie z początkowymi planami, międzynarodowy przemysł komputerowy poszedł w innym kierunku. W pracy Feigenbauma i Shrobe'a [23] można znaleźć przeglądową ocenę japońskiego projektu. Środowisko zajmujące się programowaniem w logice również podążyło dalej, w kierunku programowania relacyjnego opartego na innych technikach niż proste dopasowywanie wzorców umożliwiających na przykład operowanie na ograniczeniach liczbowych, takich jak te przedstawione na przykładzie systemu propagacji więzów z punktu 3.3.5.

a w przypadku programowania w logice — *reguły*). Główną rolę w implementacji będzie również odgrywać struktura danych ramek, która określa odpowiedniość między symbolami i przyporządkowanymi im wartościami. Dodatkowo, interesującym aspektem naszego języka zapytań będzie to, że będziemy w znaczny sposób używać strumieni wprowadzonych w rozdziale 3.

#### 4.4.1. Dedukcyjne wyszukiwanie informacji

Programowanie w logice celuje w dostarczaniu interfejsów służących do wyszukiwania informacji w bazach danych. Język zapytań, który zaimplementujemy w niniejszym rozdziale, będzie miał takie właśnie przeznaczenie.

Aby zilustrować, co robi system zapytań, pokażemy, jak można go zastosować do zarządzania bazą danych akt osobowych firmy Mikrusoft — świetnie prosperującego przedsiębiorstwa z okolic Bostonu, zajmującego się zaawansowanymi technologiami. Język ten udostępnia dane osobowe poprzez wzorce, a także potrafi wnioskować na podstawie ogólnych reguł.

#### Przykładowa baza danych

Baza danych osobowych Mikrusoftu zawiera *asercje* (stwierdzenia) dotyczące pracowników firmy. Oto informacje dotyczące Bena Bajerbita, etatowego magika komputerowego\*:

```
(address (Bajerbit Ben) (Slumerville (Ridge Road) 10))  
(job (Bajerbit Ben) (computer wizard))  
(salary (Bajerbit Ben) 60000)
```

Każda asercja to lista (tutaj trójka), której elementy same też mogą być listami.

Jako etatowy magik, Ben jest odpowiedzialny za dział komputerowy firmy i nadzoruje pracę dwóch programistów i jednego technika. Oto informacje o podwładnych Ben:

```
(address (Haker Liz P) (Cambridge (Mass Ave) 78))  
(job (Haker Liz P) (computer programmer))  
(salary (Haker Liz P) 40000)  
(supervisor (Haker Liz P) (Bajerbit Ben))  
  
(address (Tuboczny Efek) (Cambridge (Ames Street) 3))  
(job (Tuboczny Efek) (computer programmer))  
(salary (Tuboczny Efek) 35000)  
(supervisor (Tuboczny Efek) (Bajerbit Ben))
```

\* Adresy w poniższej bazie danych dowcipnie nawiązują do okolic Bostonu i Cambridge. Ze względu na ich lokalny charakter nie tłumaczymy ich tu (przyp. tłum.).

```
(address (Prawie Jan A) (Boston (Bay State Road) 22))
(job (Prawie Jan A) (computer technician))
(salary (Prawie Jan A) 25000)
(supervisor (Prawie Jan A) (Bajerbit Ben))
```

Jest również programista-praktykant nadzorowany z kolei przez Liz:

```
(address (Myślicielak Ludwik) (Slumerville (Pine Tree Road) 80))
(job (Myślicielak Ludwik) (computer programmer trainee))
(salary (Myślicielak Ludwik) 30000)
(supervisor (Myślicielak Ludwik) (Haker Liz P))
```

Wszyscy oni pracują w dziale komputerowym, o czym świadczy słowo computer występujące na początku opisu ich stanowiska (job). Ben pracuje na wysokim stanowisku. Jego zwierzchnikiem jest sam szef firmy:

```
(supervisor (Bajerbit Ben) (Walczygrosik Oliwier))
(address (Walczygrosik Oliwier) (Swellesley (Top Heap Road)))
(job (Walczygrosik Oliwier) (administration big wheel))
(salary (Walczygrosik Oliwier) 150000)
```

Oprócz działu komputerowego, którego szefem jest Ben, w firmie jest również dział księgowości, w którym pracuje główny księgowy i jego asystent:

```
(address (Scrooge Eben) (Weston (Shady Lane) 10))
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 75000)
(supervisor (Scrooge Eben) (Walczygrosik Oliwier))

(address (Cratchet Robert) (Allston (N Harvard Street) 16))
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 18000)
(supervisor (Cratchet Robert) (Scrooge Eben))
```

Jest także sekretarka szefa firmy:

```
(address (Samosia Zosia) (Slumerville (Onion Square) 5))
(job (Samosia Zosia) (administration secretary))
(salary (Samosia Zosia) 25000)
(supervisor (Samosia Zosia) (Walczygrosik Oliwier))
```

Baza danych zawiera również asercje dotyczące tego, jakie rodzaje pracy mogą być wykonywane przez osoby zajmujące się innymi rzeczami. Magik komputerowy może na przykład wykonywać zarówno pracę programisty, jak i technika komputerowego:

```
(can-do-job (computer wizard) (computer programmer))
(can-do-job (computer wizard) (computer technician))
```

Programista komputerowy może zastępować praktykanta:

```
(can-do-job (computer programmer)
             (computer programmer trainee))
```

Ponadto, jak dobrze wiadomo, sekretarka potrafi zastępować szefa:

```
(can-do-job (administration secretary)
             (administration big wheel))
```

### Zapytania proste

Język zapytań umożliwia użytkownikowi wyszukiwanie informacji w bazie danych przez wprowadzanie zapytań w odpowiedzi na znak zachęty systemu. Chcąc znaleźć na przykład wszystkich programistów, można by wprowadzić zapytanie

```
; ; Wprowadź zapytanie:
(job ?x (computer programmer))
```

System odpowiedziałby, podając następujące dane:

```
; ; Wyniki zapytania:
(job (Haker Liz P) (computer programmer))
(job (Tuboczny Efek) (computer programmer))
```

Wprowadzone zapytanie określa, że szukamy w bazie danych zapisów, które pasują do podanego wzorca. W tym przykładzie wzorzec jest dopasowywany do danych składających się z trzech elementów, z których pierwszy jest symbolem job, drugi może być czymkolwiek, a trzeci jest listą postaci (computer programmer). To „cokolwiek”, czym może być drugi element pasującej listy, jest określone we wzorcu przez zmienną x. Ogólna postać zmiennej we wzorcu to symbol (nazwa zmiennej) poprzedzony znakiem zapytania. Dalej zobaczymy, dlaczego wygodnie jest podawać nazwy zmiennych, a nie tylko wstawać do wzorca „?” na oznaczenie „czegokolwiek”. Odpowiedzią systemu na takie proste zapytanie jest wypisanie wszystkich pozycji w bazie danych, które pasują do określonego wzorca.

Wzorzec może zawierać więcej niż jedną zmienną. Na przykład zapytanie

```
(address ?x ?y)
```

spowoduje wypisanie adresów wszystkich pracowników.

Wzorzec może nie zawierać zmiennych, a wówczas zapytanie sprawdza tylko, czy w bazie danych istnieje taka pozycja. Jeśli tak, to zostanie wypisana właśnie ta pozycja; jeśli nie, nic nie zostanie wypisane.

Ta sama zmienna może pojawiać się w zapytaniu kilka razy, określając, że w każdym z tych miejsc musi pojawiać się to samo „cokolwiek”. Dlatego właśnie zmienne mają nazwy. Na przykład

```
(supervisor ?x ?x)
```

wyszukuje wszystkie osoby, które same siebie nadzorują (chociaż w naszej przykładowej bazie danych nie ma takich zapisów).

Wzorzec

```
(job ?x (computer ?type))
```

pasuje do wszystkich zapisów, których trzecim elementem jest lista dwuelementowa, której pierwszym elementem jest `computer`:

```
(job (Bajerbit Ben) (computer wizard))
(job (Haker Liz P) (computer programmer))
(job (Tuboczny Efek) (computer programmer))
(job (Prawie Jan A) (computer technician))
```

Wzorzec ten *nie* pasuje do

```
(job (Myślicielak Ludwik) (computer programmer trainee))
```

ponieważ trzeci element zapisu jest listą trójelementową, a trzeci element wzorca określa, że powinna to być lista dwuelementowa. Gdybyśmy chcieli zmienić wzorzec tak, aby trzeci element mógł być dowolną listą zaczynającą się symbolem `computer`, to moglibyśmy napisać<sup>61</sup>

```
(job ?x (computer . ?type))
```

Na przykład wzorzec

```
(computer . ?type)
```

pasuje do danych

```
(computer programmer trainee)
```

przy czym `?type` jest dopasowywane do listy `(programmer trainee)`. Pasuje on również do danych

```
(computer programmer)
```

---

<sup>61</sup> Używamy tutaj notacji kropki i ogona, wprowadzonej w ćwiczeniu 2.20.

przy czym `?type` odpowiada tu liście (`programmer`), a także pasuje do danych

`(computer)`

przy czym `?type` odpowiada tu liście pustej `()`.

Przetwarzanie prostych zapytań języka możemy opisać następująco:

- System znajduje wszystkie takie *podstawienia* zmiennych występujących w zapytaniu, które spełniają zapytanie, czyli pasują do wzorca — tzn. wszystkie takie przyporządkowania zmiennym wartości, że jeśli w miejscu zmiennych *podstawimy* odpowiadające im wartości, to uzyskamy zapis należący do bazy danych.
- System odpowiada na zapytanie, wypisując wyniki zastosowania wszystkich podstawień pasujących do tego wzorca.

Zauważmy, że jeżeli wzorzec nie zawiera zmiennych, to zapytanie sprowadza się do sprawdzenia, czy wzorzec występuje w bazie danych. Jeśli tak, to dla danej bazy danych podstawienie puste (podstawienie, które żadnej zmiennej nie przyporządkowuje żadnej wartości) pasuje do wzorca.

#### Ćwiczenie 4.55

Podaj proste zapytania, które wyszukują następujące informacje w bazie danych:

- (a) wszystkich podwładnych Bena Bajerbita;
- (b) nazwiska i stanowiska wszystkich pracowników działu księgowości;
- (c) nazwiska i adresy wszystkich pracowników mieszkających w Slumerville.

#### Zapytania złożone

Zapytania proste tworzą operacje pierwotne języka zapytań. Język zapytań udostępnia również środki łączenia umożliwiające tworzenie zapytań złożonych. To, co czyni z języka zapytań język programowania, to środki łączenia, które odzwierciedlają środki łączenia używane do formułowania wyrażeń logicznych: `and`, `or` i `not`. (Tutaj `and`, `or` i `not` nie są operacjami pierwotnymi Lispu, ale raczej operacjami wbudowanymi w język zapytań).

Możemy w następujący sposób użyć `and` do wyznaczenia adresów wszystkich programistów komputerowych:

```
(and (job ?person (computer programmer))
      (address ?person ?where))
```

Uzyskany wynik to

```
(and (job (Haker Liz P) (computer programmer))
      (address (Haker Liz P) (Cambridge (Mass Ave) 78)))
```

---

```
(and (job (Tuboczny Efek) (computer programmer))
      (address (Tuboczny Efek) (Cambridge (Ames Street) 3)))
```

Ogólnie mówiąc, zapytanie postaci

```
(and ⟨q1⟩ ⟨q2⟩ ... ⟨qn⟩)
```

spełniają wszystkie takie podstawienia zmiennych występujących w zapytaniu, które spełniają równocześnie wszystkie zapytania ⟨q<sub>1</sub>⟩ ... ⟨q<sub>n</sub>⟩.

Tak jak w przypadku zapytań prostych, system przetwarza zapytania złożone, wyznaczając wszystkie podstawienia zmiennych, które spełniają zapytanie, a następnie wyświetlając wyniki zastosowania tych podstawień do zapytania.

Inny sposób tworzenia zapytań złożonych to użycie `or`. Na przykład

```
(or (supervisor ?x (Bajerbit Ben))
    (supervisor ?x (Haker Liz P)))
```

wyszuka wszystkich pracowników podległych Benowi Bajerbitowi lub Liz P. Haker:

```
(or (supervisor (Haker Liz P) (Bajerbit Ben))
    (supervisor (Haker Liz P) (Haker Liz P)))
```

```
(or (supervisor (Tuboczny Efek) (Bajerbit Ben))
    (supervisor (Tuboczny Efek) (Haker Liz P)))
```

```
(or (supervisor (Prawie Jan A) (Bajerbit Ben))
    (supervisor (Prawie Jan A) (Haker Liz P)))
```

```
(or (supervisor (Myślicielak Ludwik) (Bajerbit Ben))
    (supervisor (Myślicielak Ludwik) (Haker Liz P)))
```

Ogólnie mówiąc, zapytanie postaci

```
(or ⟨q1⟩ ⟨q2⟩ ... ⟨qn⟩)
```

spełniają wszystkie takie podstawienia zmiennych występujących w zapytaniu, które spełniają przynajmniej jedno z zapytań ⟨q<sub>1</sub>⟩ ... ⟨q<sub>n</sub>⟩.

Zapytania złożone mogą także być budowane przy użyciu `not`. Na przykład

```
(and (supervisor ?x (Bajerbit Ben))
      (not (job ?x (computer programmer))))
```

znajduje wszystkich podwładnych Bena Bajerbita, którzy nie są programistami komputerowymi. Ogólnie mówiąc, zapytanie postaci

```
(not ⟨q1⟩)
```

spełniają wszystkie takie podstawienia zmiennych występujących w zapytaniu, które nie spełniają zapytania  $\langle q_1 \rangle$ <sup>62</sup>.

Ostatnią formą służącą do łączenia jest **lisp-value**. Jeśli **lisp-value** jest pierwszym elementem wzorca, to określa, że kolejny element jest predykatem lispowym, który należy zastosować (po zastosowaniu podstawień) do pozostałych argumentów. Ogólnie mówiąc, zapytanie postaci

```
(lisp-value <predykat> <arg1> ... <argn>)
```

spełniają wszystkie takie podstawienia zmiennych występujących w zapytaniu, dla których, po zastosowaniu ich do argumentów  $\langle arg_1 \rangle \dots \langle arg_n \rangle$ , spełniony jest  $\langle predykat \rangle$ . Chcąc znaleźć na przykład wszystkie osoby, których zarobki są większe niż 30 000 dolarów, moglibyśmy napisać<sup>63</sup>

```
(and (salary ?person ?amount)
      (lisp-value > ?amount 30000))
```

### Ćwiczenie 4.56

Sformułuj zapytania złożone, które wyszukują następujące informacje:

- nazwiska i adresy wszystkich osób podległych Benowi Bajerbitowi;
- wszystkie osoby, które zarabiają mniej niż Ben Bajerbit, wraz z ich zarobkami i zarobkami Bena Bajerbita;
- wszystkie osoby, których przełożony nie pracuje w dziale komputerowym, wraz z nazwiskiem i stanowiskiem ich przełożonego.

### Reguły

Oprócz dawania możliwości tworzenia zapytań pierwotnych i zapytań złożonych język zapytań udostępnia środki do tworzenia abstrakcji zapytań. Służą do tego *reguły*. Następująca reguła:

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
            (address ?person-2 (?town . ?rest-2))
            (not (same ?person-1 ?person-2))))
```

określa, że dwie osoby mieszkają niedaleko siebie, jeżeli mieszkają w tym samym mieście. Klauzula **not** na końcu reguły zabezpiecza przed stwierdze-

<sup>62</sup> W rzeczywistości taki opis **not** jest prawdziwy tylko dla prostych przypadków. Faktyczne działanie **not** jest bardziej skomplikowane. Osobliwym działaniem **not** zajmiemy się w punktach 4.4.2 i 4.4.3.

<sup>63</sup> Forma **lisp-value** powinna być używana wyłącznie do wykonywania operacji, które nie są udostępniane w języku zapytań. W szczególności nie powinno się jej używać do sprawdzania równości (ponieważ do tego właśnie służy mechanizm dopasowywania wzorców w języku zapytań) ani nierówności (gdyż można tego dokonać za pomocą opisanej poniżej reguły **same**).

niem, że każdy mieszka niedaleko samego siebie. Relację tożsamości `same` możemy zdefiniować za pomocą bardzo prostej reguły<sup>64</sup>:

```
(rule (same ?x ?x))
```

Następująca reguła stwierdza, że dana osoba jest w danej firmie „szychą”, jeśli jest przełożonym kogoś, kto sam też jest przełożonym:

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
           (supervisor ?x ?middle-manager)))
```

Ogólna postać reguły jest następująca:

```
(rule <nagłówek> <treść>)
```

gdzie `<nagłówek>` to wzorzec, a `<treść>` to dowolne zapytanie<sup>65</sup>. Możemy traktować regułę jak reprezentację dużego (nawet nieskończonego) zbioru asercji, a mianowicie zbioru wszystkich wyników zastosowania do nagłówka reguły podstawień, które spełniają treść reguły. Gdy opisywaliśmy zapytania proste (wzorce), powiedzieliśmy, że podstawienie zmiennych pasuje do wzorca, jeżeli po zastosowaniu takiego podstawienia do wzorca uzyskujemy asercję występującą w bazie danych. Jednak nie musi ona być zapisana wprost w bazie danych. Może to być niejawna asercja wynikająca z reguły. Na przykład w odpowiedzi na zapytanie

```
(lives-near ?x (Bajerbit Ben))
```

otrzymamy

```
(lives-near (Myślicielak Ludwik) (Bajerbit Ben))
(lives-near (Samosia Zosia) (Bajerbit Ben))
```

Chcąc znaleźć wszystkich programistów, którzy mieszkają niedaleko Beno Bajerbita, możemy wprowadzić zapytanie

```
(and (job ?x (computer programmer))
      (lives-near ?x (Bajerbit Ben)))
```

<sup>64</sup> Zwróćmy uwagę, że nie potrzebujemy `same`, aby wyrazić, że dwie rzeczy są takie same. Wystarczy, że dla każdej z nich użyjemy tej samej zmiennej — w rezultacie, przede wszystkim, zamiast o dwóch rzeczach mówimy tylko o jednej. Przyjrzyjmy się na przykład zmiennym `?town` i `?middle-manager` w poniższych regułach `lives-near` i `wheel`. Reguła `same` jest przydatna, gdy chcemy, aby dwie rzeczy były różne, tak jak `?person-1` i `?person-2` w regule `lives-near`. Choć użycie tej samej zmiennej w dwóch częściach zapytania wymusza pojawienie się w tych miejscach takiej samej wartości, jednak użycie różnych zmiennych wcale nie wymusza pojawienia się różnych wartości. (Wartości podstawiane w miejsce różnych zmiennych mogą być takie same lub różne).

<sup>65</sup> Dopuszczamy również reguły bez treści, jak w przypadku reguły `same`; przyjmujemy wówczas, że nagłówek reguły jest spełniony dla dowolnego podstawienia zmiennych.

Tak jak w przypadku procedur złożonych, możemy używać jednych reguł wewnętrz drugich reguł (jak widzieliśmy to na przykładzie `lives-near` powyżej) lub nawet definiować je rekurencyjnie. Na przykład reguła

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
          (and (supervisor ?staff-person ?middle-manager)
                (outranked-by ?middle-manager ?boss))))
```

stwierdza, że szef ma wyższe stanowisko w firmie niż pracownik, jeśli szef jest bezpośrednim przełożonym pracownika lub (rekurencyjnie) jeśli szef ma wyższe stanowisko niż bezpośredni przełożony pracownika.

#### Ćwiczenie 4.57

Zdefiniuj regułę określającą, że osoba nr 1 może zastąpić osobę nr 2, jeśli: albo osoba nr 1 wykonuje taką samą pracę co osoba nr 2, albo istnieje ktoś taki, kto wykonuje taką samą pracę co osoba nr 1 i może wykonywać pracę osoby nr 2, oraz osoba nr 1 nie jest osobą nr 2. Używając tej reguły, podaj zapytania, które wyszukują

- wszystkie osoby, które mogą zastąpić Efką Tubocznego;
- wszystkie takie osoby, które mogą zastąpić kogoś, kto zarabia więcej od tej osoby, wraz z zarobkami obydwójga.

#### Ćwiczenie 4.58

Zdefiniuj regułę określającą, że dana osoba jest „grubą rybą” w dziale, jeśli pracuje w danym dziale, ale nie ma przełożonego w tym samym dziale.

#### Ćwiczenie 4.59

Ben Bajerbit opuścił o jedno spotkanie za dużo. Bojąc się, że jego nawyk zapomnienia o spotkaniach może go kosztować posadę, Ben postanawia coś z tym zrobić. Wprowadza do bazy danych firmy Mikrusoft asercje opisujące wszystkie cotygodniowe spotkania w firmie:

```
(meeting accounting (poniedziałek 9))
(meeting administration (poniedziałek 10))
(meeting computer (środa 15))
(meeting administration (piątek 13))
```

Każda z powyższych asercji dotyczy spotkania całego działu. Ben wprowadził również zapis dotyczący spotkania pracowników firmy ze wszystkich działów (`whole-company`). Na tym spotkaniu są obecni wszyscy pracownicy firmy.

```
(meeting whole-company (środa 16))
```

- W piątek rano Ben chciałby wyciągnąć z bazy danych informacje o wszystkich spotkaniach mających miejsce tego dnia. Jakiego zapytania powinien użyć?
- Nie zrobiło to zbytniego wrażenia na Liz P. Haker. Uważa ona, że dużo bardziej przydatna byłaby możliwość dowiedzenia się, poprzez podanie jej nazwiska, o wszyst-

kich spotkaniach, w których ma wziąć udział. Tworzy ona regułę, która określa, że dana osoba bierze udział we wszystkich spotkaniach całej firmy (whole-company) oraz we wszystkich spotkaniach działu, w którym ta osoba pracuje. Uzupełnij regułę Liz, podając jej treść.

```
(rule (meeting-time ?person ?day-and-time)
      (treść reguły))
```

(c) Gdy Liz przychodzi do pracy w środę rano, zastanawia się, w jakich spotkaniach musi wziąć udział tego dnia. Mając zdefiniowaną powyższą regułę, jakie zapytanie powinna wprowadzić, aby uzyskać informację o spotkaniach.

### Ćwiczenie 4.60

Wprowadzając zapytanie

```
(lives-near ?person (Haker Liz P))
```

Liz P. Haker może ustalić, kto z jej firmy mieszka niedaleko niej i z kim mogłyby dojeżdżać razem do pracy. Jednakże gdy próbuje znaleźć wszystkie pary takich osób, które mieszkają niedaleko siebie, wprowadzając zapytanie

```
(lives-near ?person-1 ?person-2)
```

zauważa, że każda taka para osób, które mieszkają niedaleko siebie, jest wypisywana dwukrotnie; na przykład:

```
(lives-near (Haker Liz P) (Tuboczny Efek))
(lives-near (Tuboczny Efek) (Haker Liz P))
```

Dlaczego tak się dzieje? Czy istnieje sposób wyznaczenia listy par osób, które mieszkają niedaleko siebie, na której każda para występowałaby tylko raz? Odpowiedź uzasadnij.

### Logika jako programy

Spójrzmy na regułę jak na implikację logiczną: jeśli podstawienie zmiennych spełnia treść reguły, to spełnia ono również nagłówek reguły. W rezultacie możemy traktować język zapytań tak, jakby był zdolny do *logicznej dedukcji* na podstawie reguł. Jako przykład rozważmy operację *append* opisaną na początku podrozdziału 4.4. Jak stwierdziliśmy, *append* można opisać za pomocą następujących dwóch reguł:

- Dla dowolnej listy y: *append* listy pustej i y jest równe y.
- Dla dowolnych u, v, y i z: jeśli *append* list v i y jest równe z, to *append* list (*cons* u v) i y jest równe (*cons* u z).

Żeby wyrazić to w naszym języku zapytań, definiujemy dwie reguły opisujące relację

```
(append-to-form x y z)
```

która możemy interpretować jako „append list x i y jest równe z”:

```
(rule (append-to-form () ?y ?y))

(rule (append-to-form (?u . ?v) ?y (?u . ?z))
      (append-to-form ?v ?y ?z))
```

Pierwsza reguła nie ma treści, co oznacza, że nagłówek jest spełniony dla dowolnej wartości zmiennej ?y. Zwróćmy uwagę, jak w drugiej regule jest użyta notacja kropki i ogona w celu oznaczenia car i cdr listy.

Za pomocą tych dwóch reguł możemy sformułować zapytania obliczające wynik zastosowania operacji append do dwóch list:

```
;;; Wprowadź zapytanie:
(append-to-form (a b) (c d) ?z)
;;; Wyniki zapytania:
(append-to-form (a b) (c d) (a b c d))
```

Bardziej zadziwiające jest to, że tych samych reguł możemy użyć do uzyskania odpowiedzi na pytanie: „Jaka lista, po dołączeniu jej za pomocą append do listy (a b), spowoduje utworzenie listy (a b c d)?”. Odbywa się to następująco:

```
;;; Wprowadź zapytanie:
(append-to-form (a b) ?y (a b c d))
;;; Wyniki zapytania:
(append-to-form (a b) (c d) (a b c d))
```

Możemy się również spytać o wszystkie pary list, których złączenie za pomocą append daje w wyniku listę (a b c d):

```
;;; Wprowadź zapytanie:
(append-to-form ?x ?y (a b c d))
;;; Wyniki zapytania:
(append-to-form () (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) () (a b c d))
```

Może się wydawać, że system zapytań wykazuje sporo inteligencji, wywnioskując za pomocą reguł odpowiedzi na powyższe zapytania. W rzeczywistości, jak zobaczymy w następnym punkcie, system ten rozwija reguły zgodnie z dobrze określonym algorytmem. Niestety, choć system działa imponująco

w przypadku append, ogólne zasady jego działania mogą zawieść w bardziej złożonych przypadkach, jak zobaczymy w punkcie 4.4.3.

### Ćwiczenie 4.61

Następująca reguła implementuje relację next-to, która określa sąsiednie elementy listy:

```
(rule (?x next-to ?y in (?x ?y . ?u)))
(rule (?x next-to ?y in (?v . ?z))
      (?x next-to ?y in ?z))
```

Jakie będą odpowiedzi systemu na następujące zapytania:

```
(?x next-to ?y in (1 (2 3) 4))
(?x next-to 1 in (2 1 3 1))
```

### Ćwiczenie 4.62

Zdefiniuj reguły implementujące operację last-pair z ćwiczenia 2.17, której wynikiem jest lista zawierająca ostatni element niepustej listy. Przetestuj swoje reguły na zapytaniach (last-pair (3) ?x), (last-pair (1 2 3) ?x) i (last-pair (2 ?x) (3)). Czy Twoje reguły dają poprawne wyniki dla zapytań postaci (last-pair ?x (3))?

### Ćwiczenie 4.63

Następująca baza danych (zob. Księga Rodzaju 4) śledzi genealogię przodków Ady\*, poprzez Kaina aż do Adama:

```
(son Adam Kain)
(son Kain Henoch)
(son Henoch Irad)
(son Irad Mechuael)
(son Mechuael Metuszael)
(son Metuszael Lamek)
(wife Lamek Ada)
(son Ada Jabal)
(son Ada Jubal)
```

Sformułuj reguły typu „Jeśli  $S$  jest synem  $O$  i  $O$  jest synem  $D$ , to  $S$  jest wnukiem  $D$ ” oraz „Jeśli  $Z$  jest żoną  $M$  i  $S$  jest synem  $Z$ , to  $S$  jest synem  $M$ ” (co było ponoć częściej prawdą w czasach biblijnych niż dzisiaj), które pozwolą poprosić system o wyszukanie wnuka Kaina, syna Lameka i wnuków Metuszaela. (Ćwiczenie 4.69 zawiera omówienie reguł umożliwiających wnioskowanie o bardziej złożonych więzach rodzinnych).

---

\* Imiona podane zgodnie z Biblią Tysiąclecia, wyd. 2 zm., Wydawnictwo Pallotinum, 1980 (przyp. red.).

#### 4.4.2. Jak działa system zapytań

W punkcie 4.4.4 przedstawimy implementację interpretera zapytań w postaci zestawu procedur. W niniejszym punkcie prezentujemy ogólny opis struktury systemu, niezależny od niskopoziomowych szczegółów implementacji. Po tym, gdy opiszymy implementację interpretera, będziemy w stanie zrozumieć niektóre jego ograniczenia i niektóre subtelne różnice między operacjami logicznymi w języku zapytań i w logice matematycznej.

Powinno być oczywiste, że evaluator zapytań musi wykonywać pewnego rodzaju przeszukiwania w celu dopasowania zapytań do faktów i reguł zawartych w bazie danych. Można to zrealizować, implementując system zapytań jako program niedeterministyczny, używając evaluatora `amb` z podrozdziału 4.3 (zob. ćwiczenie 4.78). Inna możliwość to zrealizowanie przeszukiwania przy użyciu strumieni. Nasza implementacja będzie zgodna z tym drugim podejściem.

Serce systemu zapytań stanowią dwie operacje nazywane *dopasowywaniem wzorców* i *unifikacją*. Najpierw opiszymy dopasowywanie wzorców i wyjaśnimy, jak ta operacja, razem ze zorganizowaniem informacji w postaci strumieni ramek, pozwala na zaimplementowanie zarówno zapytań prostych, jak i zapytań złożonych. Następnie omówimy unifikację — uogólnienie dopasowywania wzorców potrzebne do zaimplementowania reguł. Na koniec pokażemy, jak wszystkie elementy interpretera zapytań składają się w jedną całość za pomocą procedury klasyfikującej wyrażenia w sposób analogiczny do tego, jak procedura `eval` klasyfikowała wyrażenia w interpreterze opisany w podrozdziale 4.1.

#### Dopasowywanie wzorców

Procedura *dopasowywania wzorców* to procedura, który sprawdza, czy określone dane pasują do określonego wzorca. Na przykład lista `((a b) c (a b))` pasuje do wzorca `(?x c ?x)` przy podstawieniu za zmienną `?x` listy `(a b)`. Ta sama lista pasuje również do wzorca `(?x ?y ?z)` przy podstawieniu za zmienne `?x` i `?z` listy `(a b)` oraz za `?y` symbolu `c`. Pasuje ona także do wzorca `((?x ?y) c (?x ?y))` przy podstawieniu `a` za `?x` i `b` za `?y`. Nie pasuje ona jednak do wzorca `(?x a ?y)`, gdyż wzorzec ten określa, że drugim elementem listy musi być symbol `a`.

Procedura dopasowywania wzorców otrzymuje od systemu zapytań jako informacje wejściowe: wzorzec, dane oraz *ramkę*\* określającą podstawienie niektórych zmiennych. Sprawdza ona, czy dane pasują do wzorca w sposób

\* Do reprezentowania podstawień używamy ramek, tak samo jak w przypadku środowisk. Powinniśmy jednak pamiętać, że tym razem ta sama struktura danych reprezentuje trochę inne pojęcie. Jak zobaczymy, ramki będą zawierać wiązania, które mogą przyporządkowywać zmiennym nie tylko konkretne wartości, ale i wzorce (przyp. tłum.).

zgodny z podstawieniem określonym przez daną ramkę. Jeśli tak, to wynikiem jej działania jest dana ramka z dołączonymi wiązaniami określonymi przez dopasowanie. W przeciwnym razie procedura sygnalizuje, że dopasowanie się nie powiodło.

Jeśli na przykład spróbujemy dopasować wzorzec ( $?x ?y ?x$ ) do listy (a b a) przy danej pustej ramce, to w wyniku otrzymamy ramkę określającą, że w miejscu  $?x$  należy podstawić a, a w miejscu  $?y$  podstawić b. Jeśli natomiast spróbujemy dopasować ten sam wzorzec do tych samych danych, ale przy danej ramce określającej, że w miejscu  $?y$  należy podstawić a, to dopasowanie się nie uda. Próbuje zaś dopasować ten sam wzorzec do tych samych danych, ale przy zadanej ramce określającej, że w miejscu  $?y$  należy podstawić b, i przy wolnej zmiennej  $?x$ , otrzymamy w wyniku daną ramkę z dołączonym wiązaniem przyporządkowującym zmiennej  $?x$  wartość a.

Procedura dopasowywania wzorców to cały mechanizm, który jest potrzebny do przetwarzania zapytań prostych nie zawierających reguł. Żeby na przykład przetworzyć zapytanie

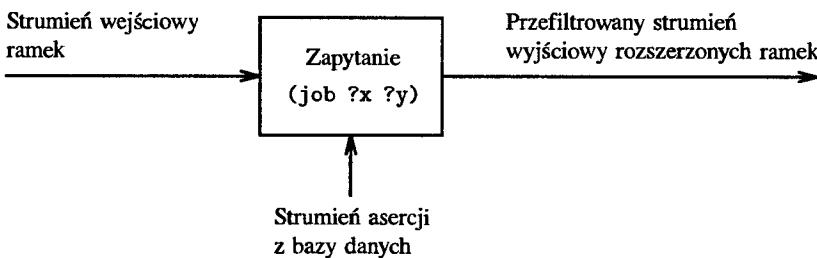
(job ?x (computer programmer))

przeglądamy wszystkie asercje w bazie danych i wybieramy te z nich, które pasują do wzorca przy zadanej początkowo pustej ramce. Dla każdego znalezionej dopasowania na podstawie otrzymanej ramki podstawiamy w miejscu zmiennej  $?x$  związaną z nią wartość.

### Strumienie ramek

Dopasowywanie elementów bazy danych do wzorców odbywa się z użyciem strumieni. Mając daną jedną ramkę, proces dopasowujący przegląda bazę danych po jednej pozycji. Dla każdej takiej pozycji proces dopasowujący generuje albo specjalny symbol, który wskazuje, że dopasowanie się nie powiodło, albo rozszerzenie danej ramki. Wyniki dla wszystkich pozycji w bazie danych są gromadzone w postaci jednego strumienia, który następnie przechodzi przez filtr odsiewający niepowodzenia. W ten sposób otrzymujemy strumień wszystkich ramek rozszerzających daną ramkę, który reprezentuje dopasowania wzorca do wybranych asercji z bazy danych<sup>66</sup>.

<sup>66</sup> Ze względu na to, że dopasowywanie jest zwykle bardzo kosztowne, warto by uniknąć wykonywania pełnego dopasowywania dla każdego elementu w bazie danych. Zwykle realizuje się to, rozbijając proces dopasowywania na szybkie, zgrubne dopasowywanie i ostateczne dopasowywanie. W trakcie zgrubnego dopasowywania odsiewa się asercje, pozostawiając jedynie niewielki zbiór kandydatów do ostatecznego dopasowania. Przy odrobinie uwagi możemy zadbać o to, aby w naszej bazie danych część zgrubnego dopasowywania była wykonywana w trakcie konstruowania bazy danych, a nie w trakcie wybierania kandydatów. Technikę taką nazywa się *indeksowaniem* bazy danych. Technologia związana z metodami indeksowania baz danych jest ogromna. Nasza implementacja, opisana w punkcie 4.4.4, realizuje naiwną postać takiego ulepszenia.



Rys. 4.4. Zapytanie przetwarzające strumień ramek

W naszym systemie zapytanie otrzymuje na wejściu strumień ramek i dla każdej ramki w tym strumieniu wykonuje opisaną powyżej operację dopasowywania, jak jest to pokazane na rys. 4.4. Oznacza to, że dla każdej ramki w strumieniu wejściowym zapytanie generuje nowy strumień złożony ze wszystkich rozszerzeń tej ramki, które dopasowują asercje z bazy danych. Następnie wszystkie te strumienie są łączone w jeden wielki strumień, który zawiera wszystkie możliwe rozszerzenia wszystkich ramek ze strumienia wejściowego. Strumień ten jest wynikiem zapytania.

Aby odpowiedzieć na zapytanie proste, używamy zapytania, które dostaje na wejściu strumień złożony z jednej pustej ramki. Powstający strumień wyjściowy zawiera wszystkie rozszerzenia pustej ramki (tzn. wszystkie odpowiedzi na dane zapytanie). Na podstawie tego strumienia generujemy następnie strumień kopii oryginalnego zapytania z wartościami z poszczególnych ramek podstawionymi w miejsce zmiennych i to ten strumień jest na koniec wypisywany.

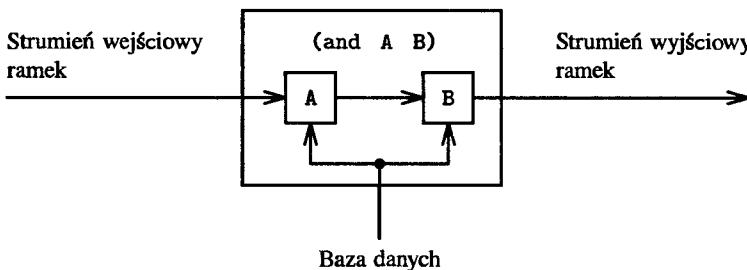
### Zapytania złożone

Cała elegancja implementacji opartej na strumieniach ramek jest widoczna dopiero w przypadku zapytań złożonych. Przy przetwarzaniu zapytań złożonych w istotny sposób wykorzystuje się to, że procedura dopasowywania wzorców wymaga, aby dopasowanie było zgodne z zadaną ramką. Chcąc na przykład obsłużyć zapytanie and złożone z dwóch zapytań, takie jak

```
(and (can-do-job ?x (computer programmer trainee))
      (job ?person ?x))
```

(nieformalnie mówiąc: „Znajdź wszystkie osoby, które mogą wykonywać pracę programisty-praktykanta”), najpierw wyznaczamy wszystkie pozycje w bazie danych, które pasują do wzorca

```
(can-do-job ?x (computer programmer trainee))
```



Rys. 4.5. Zapytanie złożone `and` jest realizowane w postaci sekwencyjnego złożenia operacji na strumieniach ramek implementujących zapytania składowe

W wyniku tego powstaje strumień ramek, z których każda zawiera podstawienie zmiennej `?x`. Następnie dla każdej takiej ramki w strumieniu znajdujemy wszystkie pozycje w bazie danych, które pasują do

`(job ?person ?x)`

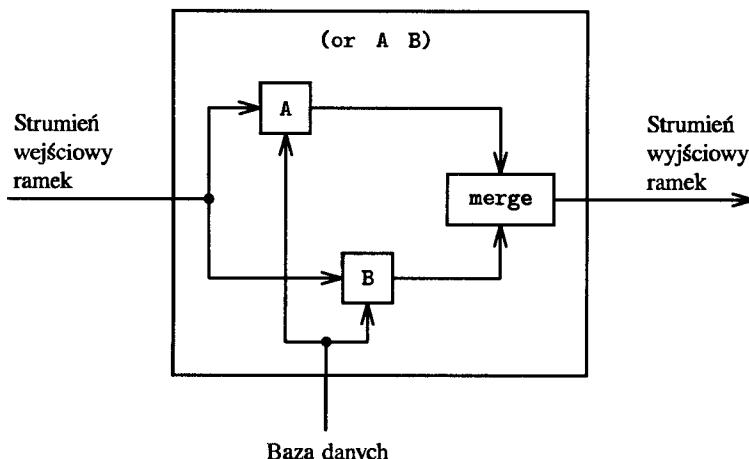
w sposób zgodny z danym podstawieniem zmiennej dla `?x`. W wyniku każdego znalezionego dopasowania powstanie ramka zawierającej podstawienie dla `?x` i `?person`. Operację `and` na dwóch zapytaniach możemy sobie przedstawić jako złożenie sekwencyjne dwóch zapytań, jak jest to pokazane na rys. 4.5. Ramki, które przechodzą przez filtr pierwszego zapytania, są filtrowane i dalej rozszerzane przez drugie zapytanie.

Na rysunku 4.6 widać analogiczną metodę obliczania zapytania `or` jako równoległego złożenia dwóch zapytań składowych. Strumień wejściowy ramek jest niezależnie przetwarzany przez oba zapytania składowe. Strumień wynikowy powstaje przez scalenie dwóch tak powstałych strumieni.

Nawet na podstawie tego ogólnego opisu widać, że przetwarzanie zapytań złożonych może być powolne. Ponieważ zapytanie może na przykład dawać w wyniku dla każdej ramki na wejściu więcej niż jedną ramkę na wyjściu, a każde zapytanie składowe w zapytaniu `and` otrzymuje na wejściu ramki generowane przez poprzednie zapytanie składowe, więc w najgorszym przypadku zapytanie `and` może wykonywać wykładniczą liczbę dopasowań ze względu na liczbę zapytań składowych (zob. ćwiczenie 4.76)<sup>67</sup>. Mimo że systemy obsługujące jedynie zapytania proste są całkiem sprawne, jednak przetwarzanie zapytań złożonych jest niesłychanie trudne<sup>68</sup>.

<sup>67</sup> Jednak taki rodzaj wykładniczej eksplozji nie występuje często w zapytaniach `and`, ponieważ zadane warunki zwykle raczej zmniejszają, a nie zwiększą liczbę przetwarzanych ramek.

<sup>68</sup> Istnieje bogata bibliografia dotycząca zarządzania systemami baz danych poświęcona głównie temu, jak efektywnie obsługiwać zapytania złożone.



Rys. 4.6. Zapytanie złożone *or* jest realizowane w postaci równoległego wykonania operacji na strumieniach ramek implementujących zapytania składowe i scalenia strumieni wynikowych

Z punktu widzenia przetwarzania strumieni ramek *not* od pewnego zapytania działa jak filtr usuwający wszystkie ramki, dla których zapytanie może być spełnione. Mając na przykład dany wzorzec

```
(not (job ?x (computer programmer)))
```

dla każdej ramki ze strumienia wejściowego próbujemy wygenerować jej rozszerzenia, które spełniają *(job ?x (computer programmer))*. Usuwamy ze strumienia wejściowego wszystkie ramki, dla których takie rozszerzenia istnieją. W wyniku otrzymujemy strumień składający się tylko z takich ramek, dla których żadne podstawienie dla *?x* nie spełnia *(job ?x (computer programmer))*. Na przykład przy przetwarzaniu zapytania

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))
```

pierwsza klauzula generuje ramki zawierające podstawienia dla *?x* i *?y*. Następnie klauzula *not* filtryuje je, usuwając wszystkie ramki, dla których podstawienie *?x* spełnia warunek, że *?x* jest programistą komputerowym<sup>69</sup>.

Forma specjalna *lisp-value* jest zaimplementowana jako podobny filtr strumienia ramek. Używamy każdej ramki ze strumienia do podstawienia wartości zmiennych we wzorcu, po czym stosujemy predykat lispowy. Usuwamy ze strumienia wszystkie te ramki, dla których predykat nie jest spełniony.

<sup>69</sup> Istnieje subtelna różnica między taką filtrową implementacją *not* a zwykłym rozumieniem negacji w logice matematycznej. Zobacz punkt 4.4.3.

## Unifikacja

Aby udostępniać w języku zapytań reguły, musimy być w stanie znajdować reguły, których nagłówki pasują do zadanych wzorców. Nagłówki procedur mają taką postać jak asercje z wyjątkiem tego, że mogą zawierać zmienne, a zatem będzie nam potrzebne uogólnienie dopasowywania wzorców — nazywane *unifikacją* — w którym zarówno „wzorzec”, jak i „dane” mogą zawierać zmienne.

Argumentami procedury unifikacji są dwa wzorce, a każdy z nich może zawierać stałe i zmienne. Procedura ta określa, czy istnieje takie podstawienie zmiennych, które po zastosowaniu do obu wzorców sprawi, że będą one sobie równe. Jeśli tak, to jej wynikiem jest ramka zawierająca takie podstawienie. Na przykład wynikiem unifikacji  $(?x \ a \ ?y)$  i  $(?y \ ?z \ a)$  będzie ramka zawierająca podstawienie, które przyporządkowuje wszystkim zmiennym  $?x$ ,  $?y$  i  $?z$  symbol  $a$ . Natomiast próba zunifikowania  $(?x \ ?y \ a)$  i  $(?x \ b \ ?y)$  nie uda się, gdyż nie istnieje taka wartość, którą można by wstawić w miejsce  $?y$ , tak aby oba wzorce były równe. (Ze względu na równość drugich elementów wzorców w miejscu  $?y$  musi być wstawione  $b$ ; równocześnie jednak, ze względu na równość trzecich elementów wzorców, w miejscu  $?y$  musi być wstawione  $a$ ). Procedura unifikacji w systemie zapytań, tak jak procedura dopasowywania wzorców, otrzymuje na wejściu ramkę i dokonuje unifikacji zgodnej z tą ramką.

Algorytm unifikacji jest najtrudniejszą technicznie częścią systemu zapytań. W wypadku wzorców złożonych może się wydawać, że wykonanie unifikacji wymaga dedukcji. Aby na przykład zunifikować  $(?x \ ?x)$  i  $((a \ ?y \ c) \ (a \ b \ ?z))$ , algorytm musi wywnioskować, że w miejscu  $?x$  powinno być podstawione  $(a \ b \ c)$ , w miejscu  $?y$  powinno być podstawione  $b$ , a w miejscu  $?z$  powinno być podstawione  $c$ . Możemy spojrzeć na ten proces jak na rozwiązywanie układu równań na składowych wzorców. Ogólnie mówiąc, mamy do czynienia z układem równań, którego rozwiązanie może wymagać pokaźnej liczby operacji<sup>70</sup>. Przykładowo, unifikację  $(?x \ ?x)$  i  $((a \ ?y \ c) \ (a \ b \ ?z))$  możemy potraktować jako określenie układu równań

$$\begin{aligned} ?x &= (a \ ?y \ c) \\ ?x &= (a \ b \ ?z) \end{aligned}$$

Z równań tych wynika, że

$$(a \ ?y \ c) = (a \ b \ ?z)$$

a z tego z kolei wynika, że

$$a = a, \ ?y = b, \ c = ?z$$

<sup>70</sup> W przypadku jednostronnego dopasowywania wzorców wszystkie równania zawierające zmienne są rozwiązywane i określają wprost wartości tych zmiennych.

a stąd w końcu wynika, że

$?x = (a \ b \ c)$

W przypadku udanego dopasowania wzorca w miejsce wszystkich zmiennych podstawiane są wartości, które zawierają jedynie stałe. Tak samo jest w przypadku wszystkich przykładów unifikacji, jakie widzieliśmy do tej pory. W ogólności jednak udana unifikacja może nie określać do końca wartości zmiennych; niektóre zmienne mogą pozostać wolne, a w miejsce innych mogą być podstawiane wartości zawierające zmienne.

Rozważmy unifikację  $(?x \ a) \ i \ ((b \ ?y) \ ?z)$ . Możemy wywnioskować, że  $?x = (b \ ?y) \ i \ a = ?z$ , ale nie możemy dokładnie stwierdzić, czym jest  $?x$  ani  $?y$ . Unifikacja nie kończy się niepowodzeniem, gdyż oczywiście można podstawić w miejsce zmiennych  $?x$  i  $?y$  takie wartości, aby oba wzorce stały się równe. Ponieważ rozważane dopasowanie w żaden sposób nie ogranicza wartości, jakie może przyjmować  $?y$ , więc żadne wiązanie zmiennej  $?y$  nie jest umieszczane w ramce. Dopasowanie to ogranicza jednak wartości, jakie mogą pojawiać się w miejscu  $?x$ . Jakakolwiek by była wartość  $?y$ ,  $?x$  musi być równe  $(b \ ?y)$ . Dlatego też do ramki jest wstawiane wiązanie, które zmiennej  $?x$  przyporządkowuje wzorzec  $(b \ ?y)$ . Jeśli wartość  $?y$  zostanie później określona i dodana do ramki (przez dopasowywanie wzorców lub unifikację, które będą musiały być zgodne z tą ramką), to poprzednie wiązanie zmiennej  $?x$  będzie odwoływać się do tej wartości<sup>71</sup>.

### **Stosowanie reguł**

Unifikacja stanowi kluczowy składnik systemu zapytań, który wnioskuje na podstawie reguł. Aby zobaczyć, jak to się odbywa, rozważmy przetwarzanie zapytania dotyczącego zastosowania reguły, takiego jak

$(lives-near ?x (Haker Liz P))$

Chcąc przetworzyć to zapytanie, stosujemy najpierw opisaną wcześniej zwykłą procedurę dopasowywania wzorców, żeby sprawdzić, czy w bazie danych są jakiekolwiek asercje pasujące do tego wzorca. (W tym przypadku nie będzie żadnych, ponieważ nasza baza danych nie zawiera żadnych bezpośrednich asercji dotyczących tego, kto mieszka niedaleko kogo). Następny krok polega na próbie unifikacji wzorca zapytania z nagłówkiem każdej z reguł. Okazuje się, że wzorzec unifikuje się z nagłówkiem reguły

<sup>71</sup> Możemy też myśleć o unifikacji w ten sposób, że jej wynikiem jest najbardziej ogólny wzorzec, który jest wspólnym uszczegółowieniem obydwu danych wzorców. Oznacza to, że wynikiem unifikacji  $(?x \ a) \ i \ ((b \ ?y) \ ?z)$  jest  $(b \ ?y) \ a$ , a wynikiem omawianej wcześniej unifikacji  $(?x \ a \ ?y) \ i \ (?y \ ?z \ a)$  jest  $(a \ a \ a)$ . Na potrzeby naszej implementacji wygodniej jest jednak myśleć o wyniku unifikacji jak o ramce, a nie o wzorcu.

```
(rule (lives-near ?person-1 ?person-2)
  (and (address ?person-1 (?town . ?rest-1))
    (address ?person-2 (?town . ?rest-2))
    (not (same ?person-1 ?person-2))))
```

w wyniku czego otrzymujemy ramkę podstawienia przyporządkowującego zmiennej `?person-2` wartość (Haker Liz P) oraz zmiennej `?x` (tę samą wartość co wartość zmiennej) `?person-1`. Następnie dla tej ramki obliczamy zapytanie złożone stanowiące treść reguły. Udane dopasowania spowodują rozszerzenie tej ramki i określenie wartości `?person-1`, a tym samym i wartości `?x`, którą możemy podstawić we wzorcu początkowego zapytania.

Ogólnie rzecz biorąc, evaluator zapytań, starając się wyznaczyć podstawienie zmiennych występujących we wzorcu, używa następującej metody dopasowywania reguł:

- zunifikuj zapytanie z nagłówkiem reguły, otrzymując w przypadku udanej unifikacji rozszerzenie zadanego podstawienia;
- oblicz zapytanie stanowiące treść reguły względem rozszerzonego podstawienia.

Zwróćmy uwagę, jak bardzo przypomina to metodę stosowania procedur w evaluatorze `eval/apply` Lispu:

- utwórz ramkę rozszerzającą środowisko procedury, wiążącą parametry z argumentami, do których procedura ma być zastosowana;
- w tak rozszerzonym środowisku oblicz wyrażenie stanowiące treść procedury.

Podobieństwo dwóch evaluatorów nie powinno nas dziwić. Tak jak definicje procedur są środkami tworzenia abstrakcji w Lispie, tak definicje reguł są środkami tworzenia abstrakcji w języku zapytań. W obu przypadkach rozwijamy abstrakcje, tworząc odpowiednie wiązania i obliczając dla nich treść reguły bądź procedury.

### Zapytania proste

Na początku niniejszego podrozdziału widzieliśmy, jak oblicza się zapytania proste nie zawierające reguł. Teraz, gdy już wiemy, jak stosować reguły, możemy opisać, jak oblicza się zapytania proste, używając zarówno reguł, jak i asercji.

Mając dany wzorzec zapytania i strumień ramek, dla każdej ramki w tym strumieniu tworzymy dwa strumienie:

- strumień ramek rozszerzonych podstawień powstały w wyniku dopasowywania wzorca (za pomocą procedury dopasowywania wzorców) do wszystkich asercji w bazie danych;
- strumień ramek rozszerzonych podstawień powstały w wyniku zastosowania (za pomocą procedury unifikacji) wszystkich reguł<sup>72</sup>.

Połączenie tych dwóch strumieni daje w wyniku strumień składający się ze wszystkich sposobów, na jakie dany wzorzec może być uzgodniony z początkową ramką. Strumienie te (po jednym dla każdej ramki w strumieniu wejściowym) są następnie łączone w jeden duży strumień zawierający wszystkie możliwe sposoby, na jakie ramki z wejściowego strumienia mogą być rozszerzane do podstawień dopasowujących dany wzorzec.

### Ewaluator zapytań i pętla sterująca

Pomimo złożoności używanych operacji dopasowywania struktura systemu przypomina w dużym stopniu strukturę evaluatora dowolnego innego języka. Procedura, która koordynuje operacje dopasowywania, nazywa się `qeval` i odgrywa ona analogiczną rolę jak procedura `eval` w przypadku Lispu. Argumentami tej procedury są zapytanie i strumień ramek. Jej wynikiem jest strumień ramek odpowiadających udanym dopasowaniom wzorca zapytania, które rozszerzają pewne ramki ze strumienia wejściowego, jak widać to na rys. 4.4. Tak jak `eval`, `qeval` rozpoznaje różne rodzaje wyrażeń (zapytań) i stosownie do tego wywołuje odpowiednią procedurę. Dla każdej formy specjalnej (`and`, `or`, `not` i `lisp-value`) oraz dla zapytań prostych mamy specjalną procedurę.

Pętla sterująca — odpowiednik procedury `driver-loop` z innych evaluatorów przedstawionych w tym rozdziale — wczytuje zapytanie z terminalu. Dla każdego wczytanego zapytania jest wywoływana procedura `qeval`, która oprócz samego zapytania otrzymuje strumień złożony z jednej pustej ramki. W wyniku powstaje strumień wszystkich możliwych podstawień (wszystkich możliwych rozszerzeń pustej ramki) spełniających zapytanie. Każde z nich jest stosowane do początkowego zapytania, w wyniku czego powstaje strumień zapytań, w których w miejscach zmiennych podstawiono wartości. Strumień ten jest następnie wypisywany<sup>73</sup>.

<sup>72</sup> Ponieważ unifikacja stanowi uogólnienie dopasowywania wzorców, moglibyśmy uprościć system i konstruować oba strumienie za pomocą procedury unifikacji. Jednakże zastosowanie prostszej procedury do prostszego przypadku ilustruje możliwości techniki dopasowywania wzorców jako takiej (w odróżnieniu od unifikacji w pełnym wydaniu).

<sup>73</sup> To, że używamy tutaj strumieni (a nie list) ramek wynika stąd, że rekurencyjne zastosowanie reguł może generować nieskończoność wiele wartości spełniających zapytanie. Odroczenie obliczeń, które ma miejsce w strumieniach, jest tutaj kluczowe — dzięki temu system wypisuje wyniki pojedynczo, w miarę ich generowania, bez względu na to, czy jest ich skończenie, czy nieskończoność wiele.

Pętla sterująca rozpoznaje również specjalne polecenie `assert!`, które sygnalizuje, że mamy do czynienia nie z zapytaniem, ale z asercją lub regułą, którą należy dodać do bazy danych. Na przykład:

```
(assert! (job (Bajerbit Ben) (computer wizard)))
(assert! (rule (wheel ?person)
  (and (supervisor ?middle-manager ?person)
    (supervisor ?x ?middle-manager))))
```

#### 4.4.3. Czy programowanie w logice to logika matematyczna?

Na pierwszy rzut oka może się wydawać, że środki łączenia używane w języku zapytań są identyczne z operacjami `and`, `or` i `not` w logice matematycznej, a stosowanie reguł języka zapytań jest tak naprawdę realizowane przez poprawną metodę wnioskowania<sup>74</sup>. Takie utożsamienie języka zapytań z logiką matematyczną nie jest jednak tak naprawdę zasadne, gdyż język zapytań określa *strukturę sterującą*, która proceduralnie interpretuje stwierdzenia logiczne. Często możemy używać tej struktury sterującej. Chcąc na przykład znaleźć wszystkich przełożonych programistów, moglibyśmy sformułować zapytanie na dwa logicznie równoważne sposoby:

```
(and (job ?x (computer programmer))
  (supervisor ?x ?y))
```

lub

```
(and (supervisor ?x ?y)
  (job ?x (computer programmer)))
```

Jeśli w firmie jest dużo więcej przełożonych niż programistów (typowy przypadek), to lepiej użyć pierwszej formy niż drugiej, gdyż przeszukujemy bazę danych dla każdego wyniku pośredniego (ramki) wygenerowanego przez pierwszą klauzulę `and`.

Celem programowania w logice jest dostarczenie programistie technik pozwalających na podzielenie problemów obliczeniowych na dwie oddzielne kwestie: „co” należy obliczyć i „jak” należy to obliczyć. Dokonuje się tego poprzez wybranie podzbioru formuł logiki matematycznej, który jest wystarczająco silny, aby opisywał wszystko, co możemy chcieć obliczyć, a jednocześnie na tyle słaby, aby miał wykonywalną interpretację proceduralną. Pomyśl polega na

<sup>74</sup> Fakt, że ta szczególna metoda wnioskowania jest poprawna, nie jest trywialny. Należy w tym celu pokazać, że z prawdziwych przesłanek można wyprowadzić jedynie prawdziwe konklusje. Metoda wnioskowania reprezentowana przez stosowanie reguł to *modus ponens*, znana metoda wnioskowania mówiąca, że jeżeli *A* jest prawdziwe oraz prawdą jest, że z *A* wynika *B*, to możemy z tego wywnioskować, że *B* jest prawdziwe.

tym, że z jednej strony program napisany w języku programowania w logice powinien faktycznie być programem, który może być wykonywany przez komputer. Sterowanie („jak” obliczać) wynika z kolejności obliczania użytej w języku. Powinniśmy umieć w takiej kolejności układać klauzule i podcele w klauzulach, aby obliczenie przebiegało w sposób, który uważa się za skuteczny i efektywny. Równocześnie powinniśmy umieć patrzeć na wyniki obliczeń (to „co” należy obliczyć) jak na proste konsekwencje wynikające z praw logiki.

Nasz język zapytań powinien być traktowany właśnie jak taki podzbiór logiki matematycznej, który można interpretować proceduralnie. Asercje reprezentują proste fakty (formuły atomowe). Reguła reprezentuje implikację — nagłówek reguły zachodzi dla tych przypadków, dla których prawdziwa jest treść reguły. Reguły mają naturalną interpretację proceduralną: aby osiągnąć nagłówek reguły, należy osiągnąć treść reguły. Zatem reguły określają obliczenia. Jednak dzięki temu, że reguły możemy traktować również jako twierdzenia logiki matematycznej, każdy „wniosek” uzyskany przez program w logice możemy uzasadnić, stwierdzając, że ten sam wynik można otrzymać, posługując się wyłącznie logiką matematyczną<sup>75</sup>.

## Pętle nieskończoności

Konsekwencją proceduralnej interpretacji programów w logice jest możliwość konstruowania, jako rozwiązań pewnych problemów, beznadziejnie nieefektywnych programów. Skrajna nieefektywność ma miejsce, gdy system, przeprowadzając wnioskowanie, wpada w pętlę nieskończoną (zapętla się). Rozważmy prosty przykład: przypuśćmy, że zakładamy bazę danych słynnych małżeństw i wprowadzamy

```
(assert! (married Mini Miki))
```

Jeśli zapytamy

```
(married Miki ?who)
```

<sup>75</sup> Musimy uściślić to stwierdzenie, przyjmując, że jeżeli mówimy o „wniosku” uzyskanym przez program w logice, to zakładamy, że obliczenie tego programu się kończy. Niestety, nawet tak uściślone stwierdzenie jest fałszywe dla naszej implementacji języka zapytań (a także dla programów napisanych w Prologu i większości pozostałych współczesnych języków programowania w logice) ze względu na użycie `not` i `lisp-value`. Jak opisujemy dalej, implementacja `not` w języku zapytań nie zawsze jest zgodna ze znaczeniem `not` w logice matematycznej. Moglibyśmy zaimplementować język zgodny z logiką matematyczną, usuwając po prostu z języka `not` i `lisp-value` i godząc się na pisanie programów jedynie przy użyciu `and` i `or`. Jednak istotnie ograniczyłoby to siłę wyrazu języka. Jednym z ważniejszych tematów badań nad programowaniem w logice jest znalezienie sposobów osiągnięcia większej zgodności z logiką matematyczną bez zbytniego poświęcania siły wyrazu.

nie otrzymamy żadnej odpowiedzi, ponieważ system nie wie, że jeżeli *A* poślubiło *B*, to *B* poślubiło *A*. Wprowadzamy zatem regułę

```
(assert! (rule (married ?x ?y)
               (married ?y ?x)))
```

i znowu pytamy

```
(married Miki ?who)
```

Niestety, to w następujący sposób spowoduje wprowadzenie systemu w pętlę nieskończoną:

- System stwierdza, że regułę `married` można zastosować; tzn. nagłówek reguły `(married ?y ?x)` można zunifikować z wzorcem `(married Miki ?who)`, otrzymując ramkę określającą, że w miejsce `?x` należy podstawić *Miki*, a w miejsce `?y` należy podstawić `?who`. Tak więc interpreter przystępuje do obliczenia dla tej ramki treści procedury: `(married ?y ?x)` — przetwarzając w rezultacie zapytanie `(married ?who Miki)`.
- W bazie danych znajduje się jedna odpowiedź podana wprost w formie asercji: `(married Mini Miki)`.
- Regułę `married` można również zastosować, a więc interpreter ponownie oblicza treść reguły, która tym razem jest równoważna `(married Miki ?who)`.

System znalazł się w pętli nieskończonej. W rzeczywistości to, czy system znajdzie prostą odpowiedź `(married Mini Miki)`, zanim wejdzie w pętlę, zależy od szczegółów implementacyjnych dotyczących kolejności, w jakiej system sprawdza elementy bazy danych. Jest to bardzo prosty przykład rodzaju pętli, jakie mogą się pojawiać. Zestawy wzajemnie powiązanych reguł mogą prowadzić do pętli, które dużo trudniej przewidzieć, a pojawienie się takiej pętli może zależeć od kolejności klauzul w zapytaniu `and` (zob. ćwiczenie 4.64) lub od niskopoziomowych szczegółów dotyczących kolejności, w jakiej system przetwarza zapytania<sup>76</sup>.

<sup>76</sup> Problem ten nie dotyczy logiki, ale proceduralnej interpretacji logiki, którą realizuje nasz interpreter. Moglibyśmy napisać interpreter, który w podanym przypadku nie wpadłby w pętlę. Móglby on na przykład wyliczać wszystkie dowody wyprowadzalne z naszych asercji i reguł, stosując przeszukiwanie wszerz zamiast przeszukiwania w głąb. Jednak w takim systemie trudniej korzystać przy konstruowaniu programów z kolejności wnioskowania. W [21] opisano próbę umieszczenia w takim programie wyszukanego sterowania. Inna technika, która nie prowadzi do tak poważnych problemów ze sterowaniem, polega na wprowadzeniu specjalnej wiedzy, takiej jak detektory szczególnych rodzajów pętli (ćwiczenie 4.67). Nie istnieje jednak niezawodny schemat zapobiegania przeglądaniu nieskończonych ścieżek w trakcie wnioskowania. Wyobraźmy sobie diaboliczną regułę postaci: „aby pokazać, że *P(x)* jest prawdą, pokaż, że *P(f(x))* jest prawdą” dla odpowiednio dobranej funkcji *f*.

## Problemy z not

Kolejna osobliwość w systemie zapytań dotyczy not. Mając daną bazę danych z punktu 4.4.1, rozważmy następujące dwa zapytania:

```
(and (supervisor ?x ?y)
      (not (job ?x (computer programmer))))
```

```
(and (not (job ?x (computer programmer)))
      (supervisor ?x ?y))
```

Te dwa zapytania dają różne wyniki. Pierwsze z nich najpierw znajduje wszystkie pozycje w bazie danych, które pasują do (supervisor ?x ?y), a następnie filtruje wygenerowane ramki, usuwając te, w których wartość ?x spełnia (job ?x (computer programmer)). Drugie zapytanie najpierw usuwa z danych wejściowych te ramki, które spełniają (job ?x (computer programmer)). Ponieważ na wejściu jest tylko jedna pusta ramka, system sprawdza, czy w bazie danych istnieje taka pozycja, która pasuje do wzorca (job ?x (computer programmer)). Z reguły w bazie danych istnieją pozycje takiej postaci, więc klauzula not odrzuca jedyną ramkę i generuje w wyniku pusty strumień ramek. W rezultacie wynikiem całego zapytania złożonego jest pusty strumień.

Problem polega na tym, że nasza implementacja not jest w rzeczywistości pomyślana jako filtr wartości zmiennych. Jeśli klauzula not jest przetwarzana dla ramki, w której pewne zmienne pozostają wolne (jak zmienna ?x w powyższym przykładzie), to wyniki działania systemu będą nieoczekiwane. Podobny problem ma miejsce, gdy używamy lisp-value — predykat lispowy nie może być obliczony, jeżeli niektóre z jego argumentów pozostają wolne. Zobacz ćwiczenie 4.77.

Istnieje też dużo poważniejsza różnica między not z języka zapytań a not z logiki matematycznej. W logice stwierdzenie „nie  $P$ ” oznacza, że  $P$  nie jest prawdziwe. Z kolei w systemie zapytań „nie  $P$ ” oznacza, że  $P$  nie da się wywnioskować z informacji zawartych w bazie danych. Mając na przykład daną bazę danych akt osobowych z punktu 4.4.1, system szczęśliwie wywnioskuje wszelkiego rodzaju stwierdzenia z not, takie jak to, że Ben Bajerbit nie jest kibicem baseballu, że na dworze nie pada i że  $2+2$  nie jest równe  $4$ <sup>77</sup>. Innymi słowy, not z języków programowania w logice odzwierciedla tzw. założenie

<sup>77</sup> Rozważmy zapytanie (not (baseball-fan (Bajerbit Ben))). System stwierdzi, że (baseball-fan (Bajerbit Ben)) nie występuje w bazie danych, więc pusta ramka nie spełnia wzorca, a co za tym idzie, nie zostanie usunięta z początkowego strumienia ramek. Wynikiem zapytania jest zatem pusta ramka, która jest używana jako podstawienie do wprowadzonego zapytania, dając w wyniku (not (baseball-fan (Bajerbit Ben))).

*o domknięciu świata, że wszystkie istotne informacje znajdują się w bazie danych*<sup>78</sup>.

### Ćwiczenie 4.64

Ludwik Myślicielak przez pomyłkę usunął z bazy danych regułę outranked-by (punkt 4.4.1). Gdy zdał sobie z tego sprawę, szybko ją ponownie wprowadził. Niestety, dokonał drobnej zmiany w regule i zapisał ją jako

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
          (and (outranked-by ?middle-manager ?boss)
                (supervisor ?staff-person ?middle-manager))))
```

Zaraz po tym, jak Ludwik wprowadził tę informację do systemu, Zosia Samosia postanawia sprawdzić, kto ma wyższe stanowisko niż Ben Bajerbit. Wprowadza w tym celu zapytanie

```
(outranked-by (Bajerbit Ben) ?who)
```

Po wypisaniu odpowiedzi system się zapętla. Wyjaśnij, dlaczego tak się dzieje?

### Ćwiczenie 4.65

Efek Tuboczny, czekając z utęsknieniem na dzień, gdy awansuje w firmie, wprowadził zapytanie, które wyszukuje wszystkie szachy (za pomocą reguły wheel z punktu 4.4.1):

```
(wheel ?who)
```

Ku jego zaskoczeniu system odpowiedział, wypisując

```
; ; Wyniki zapytania:
(wheel (Walczygrosik Oliwier))
(wheel (Bajerbit Ben))
(wheel (Walczygrosik Oliwier))
(wheel (Walczygrosik Oliwier))
(wheel (Walczygrosik Oliwier))
```

Dlaczego Oliwier Walczygrosik jest wymieniony czterokrotnie?

### Ćwiczenie 4.66

Ben uogólnia system zapytań, aby umożliwić badanie statystyk dotyczących przedsiębiorstwa. Na przykład w celu wyznaczenia łącznych zarobków wszystkich programistów komputerowych będzie można napisać

```
(sum ?amount
      (and (job ?x (computer programmer))
            (salary ?x ?amount)))
```

<sup>78</sup> Omówienie i uzasadnienie takiego traktowania not można znaleźć w artykule Clarka [12].

Ogólnie rzecz biorąc, nowy system Ben'a dopuszcza wyrażenia postaci

```
(accumulation-function <zmienna>
  < wzorzec zapytania >)
```

gdzie accumulation-function może być czymś takim, jak sum (suma), average (średnia) czy maximum (maksimum). Ben uważa, że zrobienie tego to pestka. Chce on po prostu przekazać wzorzec zapytania do qeval. W wyniku powstanie strumień ramek. Ben chce następnie przepuścić ten strumień przez funkcję przekształcającą, która z każdej ramki w strumieniu wydobywa wartość określonej zmiennej, po czym tak powstały strumień wartości ma być przekazywany do funkcji kumulującej. Kiedy Ben kończy implementację i ma zamiar ją wypróbować, przychodzi Efek, który cały czas głowił się nad wynikami zapytania wheel z ćwiczenia 4.65. Gdy Efek pokazuje Benowi odpowiedź systemu, ten łapie się za głowę i jęczy: „O nie, mój prosty schemat kumulacji wartości nie będzie działał!”.

Co Ben właśnie sobie uświadomił? Naszkicuj metodę, jakiej może użyć Ben, aby wyjść z zaistniałej sytuacji.

### Ćwiczenie 4.67

Obmyśl, w jaki sposób można by zainstalować w systemie zapytań wykrywacz zapętleń, który pozwoliłby uniknąć prostych zapętleń, przedstawionych w tekście i w ćwiczeniu 4.64. Ogólny pomysł polega na tym, że system powinien pamiętać coś w rodzaju historii aktualnego łańcucha wniosków i nie powinien przystępować do przetwarzania zapytań, nad którymi właśnie pracuje. Opisz, jakiego rodzaju informacje (wzorce i ramki) powinny należeć do tej historii i jak powinno się odbywać takie sprawdzanie. (Po tym, jak w punkcie 4.4.4 zapoznasz się ze szczegółami implementacji systemu zapytań, możesz chcieć zmodyfikować system tak, aby zawierał wykrywacz zapętleń).

### Ćwiczenie 4.68

Zdefiniuj regułę implementującą operację reverse z ćwiczenia 2.18, której wynikiem jest lista złożona z tych samych elementów co dana lista, ale w odwrotnej kolejności. (Wskazówka: Użyj append-to-form). Czy Twój reguła poradzi sobie zarówno z (reverse (1 2 3) ?x), jak i (reverse ?x (1 2 3))?

### Ćwiczenie 4.69

Mając daną na początku bazę danych i reguły sformułowane w ćwiczeniu 4.63, obmyśl regułę pozwalającą dodawać „pra-” (ang. great-) do relacji między wnukiem i dziadkiem. Powinna ona umożliwić systemowi wywnioskowanie, że Irad jest prawnukiem Adama, a Jabal i Jubal są prapraprapraprawnukami Adama. (Wskazówka: Fakt dotyczący Irada powinien być na przykład reprezentowany jako ((great grandson) Adam Irad). Napisz regułę, która określa, czy lista kończy się słowem grandson (wnuk). Użyj jej do wyrażenia reguły, która pozwala wywnioskować relację ((great . ?rel) ?x ?y), gdzie ?rel jest listą zakończoną słowem grandson). Sprawdź swoje reguły na takich zapytaniach, jak ((great grandson) ?g ?ggs) bądź (?relationship Adam Irad).

#### 4.4.4. Implementacja systemu zapytań

W punkcie 4.4.2 opisaliśmy działanie systemu zapytań. Uzupełnimy teraz szczegóły i przedstawimy pełną implementację systemu.

##### 4.4.4.1. Pętla sterująca i podstawienia

Pętla sterująca systemu zapytań wielokrotnie wczytuje wyrażenia wejściowe. Jeśli wyrażenie jest regułą lub asercją, którą należy dodać do bazy danych, to informacja ta jest dodawana. W przeciwnym razie zakłada się, że wyrażenie jest zapytaniem. Pętla przekazuje to zapytanie do evaluatora `qeval` wraz z początkowym strumieniem ramek złożonym z jednej pustej ramki. W wyniku obliczeń powstaje strumień ramek reprezentujących podstawienia spełniające zapytanie, przyporządkowane zmiennym wartości znalezione w bazie danych. Na podstawie tych ramek jest tworzony nowy strumień złożony z kopii początkowego zapytania, w którym w miejsce zmiennych podstawiono wartości określone przez strumień ramek, i ten strumień jest w końcu wypisywany na terminalu:

```
(define input-prompt ";;; Wprowadź zapytanie:")
(define output-prompt ";;; Wyniki zapytania:")

(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline)
           (display "Asercja dodana do bazy danych.")
           (query-driver-loop))
          (else
           (newline)
           (display output-prompt)
           (display-stream
             (stream-map
               (lambda (frame)
                 (instantiate q
                               frame
                               (lambda (v f)
                                 (contract-question-mark v))))
               (qevel q (singleton-stream '())))))
           (query-driver-loop)))))
```

Używamy tutaj, tak jak i w innych evaluatorach przedstawionych w niniejszym rozdziale, składni abstrakcyjnej wyrażeń języka zapytań. Implementacja składni wyrażeń, łącznie z predykatem `assertion-to-be-added?` i selektorem

`add-assertion-body`, jest przedstawiona w podpunkcie 4.4.4.7. Procedura `add-rule-or-assertion!` jest zdefiniowana w podpunkcie 4.4.4.5.

Przed przystąpieniem do przetwarzania danego wyrażenia pętla sterująca przekształca je składowo na postać, którą można przetwarzać efektywniej. Obejmuje to zmianę reprezentacji zmiennych we wzorcu. W trakcie podstawiania zmiennych w zapytaniu wszystkie zmienne, które pozostają wolne, są przed wypisaniem przekształcane z powrotem na postać wejściową. Przekształcenia te są wykonywane przez dwie procedury `query-syntax-process` i `contract-question-mark` (podpunkt 4.4.4.7).

Przekształcając wyrażenie zgodnie z podstawieniem określonym przez daną ramkę, kopujemy wyrażenie i podstawiamy w miejsce zmiennych odpowiadające im wartości. Do tych wartości również stosujemy podstawienie, gdyż mogą one zawierać zmienne (np. gdy w miejscu zmiennej `?x` w `exp` podstawiane jest `?y`, a w miejscu `?y`, w wyniku unifikacji, podstawiane jest 5). Akcja, którą należy podjąć, gdy w miejscu zmiennej nie można nic podstawić, jest określona przez argument proceduralny `instantiate`.

```
(define (instantiate exp frame unbound-var-handler)
  (define (copy exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
             (if binding
                 (copy (binding-value binding))
                 (unbound-var-handler exp frame))))
           ((pair? exp)
            (cons (copy (car exp)) (copy (cdr exp))))
            (else exp)))
    (copy exp))
```

Procedury, które operują na podstawieniach, są zdefiniowane w podpunkcie 4.4.4.8.

#### 4.4.4.2. Evaluator

Procedura `qeval`, wywoływana przez `query-driver-loop`, stanowi podstawowy evaluator w systemie zapytań. Otrzymuje ona na wejściu zapytanie oraz strumień ramek, a jej wynikiem jest strumień rozszerzonych ramek. Rozpoznaje ona formy specjalne poprzez rozdzielanie sterowane danymi przy użyciu procedur `get` i `put`, tak jak to robiliśmy, implementując operacje ogólne w rozdziale 2. Każde zapytanie, które nie zostanie rozpoznane jako forma specjalna, jest traktowane jako zapytanie proste i przetwarzane za pomocą procedury `simple-query`.

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
```

```
(if qproc
  (qproc (contents query) frame-stream)
  (simple-query query frame-stream)))
```

Procedury `type` i `contents`, zdefiniowane w podpunkcie 4.4.4.7, implementują składnię abstrakcyjną form specjalnych.

### Zapytania proste

Procedura `simple-query` przetwarza zapytania proste. Jej argumentami są zapytanie proste (wzorzec) oraz strumień ramek, a jej wynikiem jest strumień ramek powstały w wyniku rozszerzenia każdej ramki przez wszystkie możliwe jej dopasowania w bazie danych.

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (stream-append-delayed
        (find-assertions query-pattern frame)
        (delay (apply-rules query-pattern frame))))
    frame-stream))
```

Dla każdej ramki w strumieniu wejściowym: w celu dopasowania wzorca do wszystkich asercji w bazie danych używamy procedury `find-assertions` (podpunkt 4.4.4.3), w wyniku czego powstaje strumień rozszerzonych ramek, a w celu zastosowania wszystkich możliwych reguł używamy procedury `apply-rules` (podpunkt 4.4.4.4), w wyniku czego powstaje drugi strumień rozszerzonych ramek. Te dwa strumienie ramek łączymy (za pomocą `stream-append-delayed`, podpunkt 4.4.4.6), tworząc strumień wszystkich możliwych sposobów, na jakie dany wzorzec może być spełniony zgodnie z początkową ramką (zob. ćwiczenie 4.71). Strumienie odpowiadające poszczególnym ramkom są łączone za pomocą procedury `stream-flatmap` (podpunkt 4.4.4.6), tworząc jeden duży strumień wszystkich sposobów, na jakie ramki z początkowego strumienia wejściowego mogą być rozszerzone tak, aby spełniały zadany wzorzec.

### Zapytania złożone

Zapytania `and` są obsługiwane w sposób przedstawiony na rys. 4.5 za pomocą procedury `conjoin`. Argumentami tej procedury są podwyrażenia koniunkcji oraz strumień rozszerzonych ramek, a jej wynikiem jest strumień rozszerzonych ramek. Najpierw `conjoin` przetwarza strumień ramek, wyznaczając strumień wszystkich możliwych rozszerzeń ramek, które spełniają pierwsze podwyrażenie koniunkcji. Następnie ten nowy strumień ramek jest rekurencyjnie przetwarzany przez procedurę `conjoin` dla pozostałych podwyrażeń.

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjunctions conjuncts)
                (qeval (first-conjunction conjuncts)
                       frame-stream))))
```

### Wyrażenie

```
(put 'and 'qeval conjoin)
```

instaluje w qeval procedurę conjoin tak, że kierowane są do niej napotkane formy and.

Zapytania or są obsługiwane podobnie, jak widać to na rys. 4.6. Strumienie wynikowe tworzone dla różnych podwyrażeń alternatywy or są obliczane niezależnie, a następnie scalane za pomocą procedury `interleave-delayed` z podpunktu 4.4.4.6. (Zobacz ćwiczenia 4.71 i 4.72).

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
        (qeval (first-disjunct disjuncts) frame-stream)
        (delay (disjoin (rest-disjuncts disjuncts)
                        frame-stream))))))
```

```
(put 'or 'qeval disjoin)
```

Predykaty i selektory dla składni koniunkcji i alternatywy są podane w podpunkcie 4.4.4.7.

### Filtры

Filtr not jest obsługiwany zgodnie z metodą naszkicowaną w punkcie 4.4.2. Każdą ramkę w strumieniu wejściowym staramy się tak rozszerzyć, aby spełniała zanegowane zapytanie. Do strumienia wyjściowego przechodzą jedynie te ramki, których nie dało się rozszerzyć.

```
(define (negate operands frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (stream-null? (qeval (negated-query operands)
                                (singleton-stream frame)))
          (singleton-stream frame)
          the-empty-stream)))
    frame-stream))

(put 'not 'qeval negate)
```

**Lisp-value** stanowi filtr podobny do **not**. Każda ramka w strumieniu jest używana do podstawienia zmiennych we wzorcu, po czym jest stosowany podany predykat i te ramki, dla których nie jest on spełniony, są usuwane ze strumienia. Jeśli we wzorcu pozostają zmienne wolne, to zgłaszany jest błąd.

```
(define (lisp-value call frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (execute
            (instantiate
              call
              frame
              (lambda (v f)
                (error "Nieznana zmienna -- LISP-VALUE" v))))
            (singleton-stream frame)
            the-empty-stream))
        frame-stream)))
(put 'lisp-value 'qeval lisp-value)
```

Procedura **execute**, która stosuje predykat do argumentów, musi obliczyć (za pomocą **eval**) wartość wyrażenia stanowiącego predykat, aby otrzymać procedurę, którą należy zastosować. Nie może ona jednak obliczać wartości argumentów, gdyż są to już faktyczne wartości argumentów, a nie wyrażenia, których obliczenie (w Lispie) dałoby w wyniku wartości argumentów. Zwróćmy uwagę, że procedura **execute** jest zaimplementowana za pomocą procedur **eval** i **apply** z działającego pod spodem systemu lispowego.

```
(define (execute exp)
  (apply (eval (predicate exp) user-initial-environment)
        (args exp)))
```

Forma specjalna **always-true** umożliwia formułowanie zapytań, które są zawsze spełnione. Ignoruje ona swoją treść (zwykle pustą) i przekazuje po prostu wszystkie ramki ze strumienia wejściowego do wyjściowego. **Always-true** jest używana przez selektor **rule-body** (podpunkt 4.4.4.7) do określenia treści reguł, które zostały zdefiniowane bez treści (tzn. reguł, których nagłówki są zawsze spełnione).

```
(define (always-true ignore frame-stream) frame-stream)
(put 'always-true 'qeval always-true)
```

Selektory definiujące składnię **not** i **lisp-value** są podane w podpunkcie 4.4.4.7.

#### 4.4.4.3. Wyszukiwanie asercji za pomocą dopasowywania wzorców

Argumentami procedury `find-assertions`, wywoływanej przez `simple-query` (podpunkt 4.4.4.2), są wzorzec oraz ramka. Jej wynikiem jest strumień ramek, a każda z nich rozszerza daną ramkę, dopasowując ją do zawartości bazy danych. Używa ona procedury `fetch-assertions` (podpunkt 4.4.4.5) do uzyskania strumienia wszystkich asercji w bazie danych, które należy spróbować dopasować do danego wzorca i ramki. Procedura `fetch-assertions` została tu wprowadzona dlatego, że często można zastosować proste testy, które eliminują z puli kandydatów do udanego dopasowania wiele pozycji z bazy danych. System nadal by działał, gdybyśmy wyeliminowali `fetch-assertions` i po prostu sprawdzali strumień wszystkich asercji w bazie danych, ale wówczas obliczenie byłoby mniej efektywne, gdyż procedura dopasowująca byłaby wywoływana dużo częściej.

```
(define (find-assertions pattern frame)
  (stream-flatmap (lambda (datum)
    (check-an-assertion datum pattern frame))
    (fetch-assertions pattern frame)))
```

Argumentami procedury `check-an-assertion` są wzorzec, obiekt danych (asercja) oraz ramka, a jej wynikiem jest albo jednoelementowy strumień zawierający rozszerzoną ramkę, albo pusty strumień (`the-empty-stream`), jeśli dopasowanie się nie udało.

```
(define (check-an-assertion assertion query-pat query-frame)
  (let ((match-result
        (pattern-match query-pat assertion query-frame)))
    (if (eq? match-result 'failed)
        the-empty-stream
        (singleton-stream match-result))))
```

Wynikiem podstawowej procedury dopasowywania wzorców jest albo symbol `failed` (nie udało się), albo rozszerzenie danej ramki. Zasadniczy pomysł dopasowywania polega na tym, aby dopasowywać wzorzec do danych, element po elemencie, kumulując podstawienia zmiennych występujących we wzorcu. Gdy wzorzec i obiekt danych są tym samym, dopasowanie udaje się i przekazywana jest ramka zawierająca skumulowane podstawienie. W przeciwnym razie, jeśli wzorzec jest zmienną, to rozszerzamy bieżącą ramkę, przyporządkowując zmiennej dane tak długo, jak długo jest to zgodne z podstawieniami już zgromadzonymi w ramce. Jeśli zaś wzorzec i dane są parami, to (rekurencyjnie) dopasowujemy `car` wzorca do `car` danych, otrzymując w wyniku ramkę; dla tej ramki dopasowujemy następnie `cdr` wzorca do `cdr` danych. Jeśli żaden z powyższych przypadków nie zachodzi, dopasowanie nie jest możliwe i jako wynik przekazywany jest symbol `failed`.

```
(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat)
                       (cdr dat)
                       (pattern-match (car pat)
                                     (car dat)
                                     frame)))
        (else 'failed)))
```

Oto procedura rozszerzająca ramkę przez dodanie nowego wiązania zmiennej, o ile jest ono zgodne z wiązaniemami już zawartymi w ramce:

```
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
        (pattern-match (binding-value binding) dat frame)
        (extend var dat frame))))
```

Jeśli w ramce nie ma żadnego wiązania danej zmiennej, to dodajemy wiązanie, które przyporządkowuje zmiennej dane. W przeciwnym razie dopasowujemy (w ramce) wartość podstawianą w miejsce danej zmiennej do danych. Jeżeli pamiętała w ramce wartość podstawiana w miejsce zmiennej zawiera tylko stałe, a tak musi być, jeśli została ona zapamiętana przez `extend-if-consistent` w ramach dopasowywania wzorców, to procedura dopasowywania sprawdza po prostu, czy zapamiętana i nowa wartość są takie same. Jeśli tak, to jej wynikiem jest niezmieniona ramka; jeśli nie, to wynikiem jest znacznik niepowodzenia. Jeśli jednak zapamiętaną wartość wstawiono do ramki w ramach unifikacji (zob. podpunkt 4.4.4.4), to może zawierać zmienne. Rekurencyjne dopasowanie zapamiętanego wzorca do nowych danych powoduje sprawdzenie podstawienia zmiennych występujących we wzorcu i/lub dodanie dla nich nowych wiązań. Przypuśćmy na przykład, że mamy ramkę, w której `?x` jest związane z `(f ?y)`, a `?y` jest wolne, i że chcemy powiększyć tę ramkę, dodając do niej wiązanie przyporządkowujące zmiennej `?x` wartość `(f b)`. Sprawdzamy zmienną `?x` i okazuje się, że jest ona związana z `(f ?y)`. Prowadzi to nas do próby dopasowania w tej samej ramce `(f ?y)` do zaproponowanej nowej wartości `(f b)`. Ostatecznie dopasowanie to rozszerza ramkę, wiążąc `?y` z `b`. Zmienna `?x` pozostaje związana z `(f ?y)`. Nigdy nie modyfikujemy zapamiętanych wiązań i nigdy nie zapamiętujemy więcej niż jednego wiązania dla danej zmiennej.

Procedury używane przez `extend-if-consistent` do operowania na wiązaniach są zdefiniowane w podpunkcie 4.4.4.8.

### Notacja kropki i ogona we wzorcach

Jeśli wzorzec zawiera kropkę, po której występuje zmienna, to zmienna ta jest dopasowywana do pozostałej części listy danych (a nie tylko do następnego elementu na liście danych), tak jak można się tego spodziewać zgodnie z notacją kropki i ogona, opisaną w ćwiczeniu 2.20. Choć procedura dopasowywania wzorców, którą właśnie zaimplementowaliśmy, nie zwraca uwagi na kropki, jednak działa tak, jak byśmy tego chcieli. Dzieje się tak dlatego, że operacja pierwotna `read` Lispu, której używamy w `query-driver-loop` do wczytywania zapytania i reprezentowania go jako struktury listowej, traktuje kropkę w specjalny sposób.

Gdy `read` napotyka kropkę, zamiast przyjmować, że następny element jest kolejnym elementem listy (tzn. carem `consa`, którego `cdr` będzie pozostałą częścią listy), przyjmuje, że następny element będzie `cdrem` struktury listowej. Na przykład strukturę listową będącą wynikiem `read` dla wzorca `(computer ?type)` można otrzymać, obliczając wyrażenie `(cons 'computer (cons '?type '()))`, a taką, która jest wynikiem wczytania wzorca `(computer . ?type)`, można skonstruować, obliczając wyrażenie `(cons 'computer '?type)`.

W ten sposób pattern-match, rekurencyjnie porównując `cary` i `cdry` listy danych i wzorca z kropką, dopasowuje w końcu zmienną po kropce (która stanowi `cdr` wzorca) do podlisty listy danych, wiążąc zmienną z tą podlistą. Dopasowując na przykład wzorzec `(computer . ?type)` do `(computer programmer trainee)`, dopasujemy `?type` do listy `(programmer trainee)`.

#### 4.4.4.4. Reguły i unifikacja

Procedura `apply-rules` jest odpowiednikiem `find-assertions` (podpunkt 4.4.4.3) dla reguł. Jej argumentami są wzorzec i ramka. Stosując reguły z bazy danych, tworzy ona strumień rozszerzających ramek. Do każdej reguły ze strumienia reguł, które mogą mieć zastosowanie `apply-a-rule` (wybranych za pomocą `fetch-rules`, podpunkt 4.4.4.5), stosujemy, za pomocą `stream-flatmap`, procedurę `apply-a-rule`, a następnie łączymy uzyskane strumienie ramek.

```
(define (apply-rules pattern frame)
  (stream-flatmap (lambda (rule)
    (apply-a-rule rule pattern frame))
    (fetch-rules pattern frame)))
```

Procedura `apply-a-rule` stosuje reguły zgodnie z metodą naszkicowaną w punkcie 4.4.2. Najpierw powiększa ramkę przekazaną jako argument, unifikując dla danej ramki nagłówek reguły ze wzorcem. Jeśli unifikacja się udaje, obliczana jest treść reguły dla nowo powstałej ramki.

Zanim jednak to wszystko będzie miało miejsce, program przemianowuje wszystkie zmienne w regule, nadając im nowe, niepowtarzalne nazwy. Ma to zapobiec mylению ze sobą zmiennych pochodzących z różnych zastosowań reguł. Jeśli na przykład dwie reguły używają zmiennej o nazwie  $?x$ , to każda z tych reguł, gdy zostanie zastosowana, może wprowadzić do ramki wiązanie dla zmiennej  $?x$ . Jednak te dwie zmienne  $?x$  nie mają ze sobą nic wspólnego i nie powinniśmy dać się nabrać, że ich wiązania muszą być ze sobą zgodne. Zamiast przemianowywać zmienne, moglibyśmy obmyślić lepszą strukturę środowisk; jednak podejście polegające na przemianowywaniu, które tutaj wybraliśmy jest najprostsze, nawet jeśli nie jest najefektywniejsze. (Zobacz ćwiczenie 4.79).

Oto procedura `apply-a-rule`:

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (let ((unify-result
           (unify-match query-pattern
                         (conclusion clean-rule)
                         query-frame)))
      (if (eq? unify-result 'failed)
          the-empty-stream
          (qeval (rule-body clean-rule)
                 (singleton-stream unify-result)))))))
```

Selektory `rule-body` i `conclusion`, wydzielające składowe reguł, są zdefiniowane w podpunkcie 4.4.4.7.

Niepowtarzalne nazwy zmiennych generujemy, kojarząc z każdym zastosowaniem reguły niepowtarzalny identyfikator (taki jak liczba) i łącząc ten identyfikator z pierwotnymi nazwami zmiennych. Jeśli na przykład identyfikatorem zastosowania reguły jest 7, to możemy zmienić w regule każde  $?x$  na  $?x-7$ , a każde  $?y$  na  $?y-7$ . (Procedury `make-new-variable` i `new-rule-application-id` są zamieszczone razem z procedurami składowymi w podpunkcie 4.4.4.7).

```
(define (rename-variables-in rule)
  (let ((rule-application-id (new-rule-application-id)))
    (define (tree-walk exp)
      (cond ((var? exp)
             (make-new-variable exp rule-application-id))
            ((pair? exp)
             (cons (tree-walk (car exp))
                   (tree-walk (cdr exp))))
            (else exp)))
    (tree-walk rule)))
```

Algorytm unifikacji jest zaimplementowany w postaci procedury, która otrzymuje jako argumenty dwa wzorce i ramkę, a daje w wyniku albo rozszerzoną ramkę, albo symbol `failed`. Procedura unifikacji przypomina procedurę dopasowywania wzorców z wyjątkiem tego, że jest symetryczna — zmienne mogą się pojawiać po obu dopasowywanych stronach. Procedura `unify-match` jest zasadniczo taka sama jak `pattern-match` z wyjątkiem dodatkowego wiersza w kodzie (zaznaczonego poniżej przez „\*\*\*”), który dotyczy przypadku, gdy obiekt po prawej stronie dopasowania jest zmienną.

```
(define (unify-match p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((var? p2) (extend-if-possible p2 p1 frame)) ; ***
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
                      (cdr p2)
                      (unify-match (car p1)
                                   (car p2)
                                   frame)))
        (else 'failed)))
```

W trakcie unifikacji, tak jak w jednostronnym dopasowywaniu wzorców, chcemy przyjąć zaproponowane rozszerzenie ramki tylko wtedy, gdy jest ono zgodne z istniejącymi wiązaniami. Procedura `extend-if-possible`, której używamy w unifikacji, jest taka sama jak `extend-if-consistent` używana przy dopasowywaniu wzorców z wyjątkiem dwóch specjalnych sprawdzeń zaznaczonych w poniższym programie przez „\*\*\*”. W pierwszym z rozważanych przypadków, jeśli zmienna, którą staramy się dopasować, jest wolna, ale wartość, z którą staramy się ją zunifikować, sama jest (inną) zmienną, to musimy sprawdzić, czy jest ona związana, i jeśli tak, dopasować daną zmienną do związanego z nią wartości. Jeśli obie strony dopasowania są wolne, to możemy związać dowolną z nich z drugą.

Drugi przypadek dotyczy próby związania zmiennej ze wzorcem, który zawiera tę zmienną. Sytuacja taka może się pojawić, ilekroć zmienna powtarza się w obu wzorcach. Rozważmy na przykład unifikację dwóch wzorców  $(?x ?x)$  i  $(?y \langle \text{wyrażenie zawierające } ?y \rangle)$  dla ramki, w której obie zmienne  $?x$  i  $?y$  są wolne. Najpierw  $?x$  jest dopasowywane do  $?y$ , w wyniku czego z  $?x$  związywany jest  $?y$ . Następnie ta sama zmienna  $?x$  jest dopasowywana do danego wyrażenia zawierającego  $?y$ . Ponieważ  $?x$  jest już związane z  $?y$ , spowoduje to próbę dopasowania  $?y$  do wyrażenia zawierającego  $?y$ . Jeśli spojrzymy na unifikację jak na wyznaczenie wartości zmiennych występujących we wzorcach, po podstawieniu których wzorce są takie same, to wzorce te określają, że należy znaleźć takie  $?y$ , które jest równe wyrażeniu zawierającemu

cemu  $?y$ . Nie istnieje ogólna metoda rozwiązywania takich równań, w związku z czym odrzucamy takie wiązania — przypadki te są rozpoznawane za pomocą predykatu `depends-on?`<sup>79</sup>. Jednak nie chcemy odrzucać prób zunifikowania zmiennej z samą sobą. Rozważmy na przykład unifikację  $(?x ?x)$  i  $(?y ?y)$ . Przy drugiej próbie związania  $?x$  z  $?y$  dopasowujemy  $?y$  (zapamiętaną wartość  $?x$ ) do  $?y$  (nowej wartości  $?x$ ). Zajmuje się tym klauzula `equal?` w procedurze `unify-match`.

```
(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
            (unify-match
              (binding-value binding) val frame))
          ((var? val) ; ***
           (let ((binding (binding-in-frame val frame)))
             (if binding
                 (unify-match
                   var (binding-value binding) frame)
                 (extend var val frame))))
          ((depends-on? val var frame) ; ***
           'failed)
          (else (extend var val frame)))))
```

<sup>79</sup> W ogólności do zunifikowania  $?y$  z wyrażeniem zawierającym  $?y$  wymagane jest znalezienie punktu stałego równania  $?y = \langle \text{wyrażenie zawierające } ?y \rangle$ . Czasami można składniowo sformułować wyrażenie, które wygląda na rozwiązywanie. Wydaje się na przykład, że punktem stałym  $?y = (f ?y)$  jest  $(f (f (f \dots)))$ , co możemy uzyskać, wychodząc od  $(f ?y)$  i wciąż podstawiając  $(f ?y)$  w miejsce  $?y$ . Niestety, nie każde takie równanie ma sensowny punkt stały. Kwestie, z którymi mamy tu do czynienia, są podobne do kwestii związanych z operowaniem na szeregach nieskończonych w matematyce. Wiadomo na przykład, że liczba 2 jest rozwiązaniem równania  $y = 1 + y/2$ . Wychodząc od wyrażenia  $1 + y/2$  i podstawiając wciąż  $1 + y/2$  w miejsce  $y$ , otrzymujemy

$$2 = y = 1 + y/2 = 1 + (1 + y/2)/2 = 1 + 1/2 + y/4 = \dots$$

co rozwija się w szereg

$$2 = 1 + 1/2 + 1/4 + 1/8 + \dots$$

Jeśli jednak spróbujemy zrobić to samo, wychodząc od stwierdzenia, że  $-1$  jest rozwiązaniem równania  $y = 1 + 2y$ , to otrzymamy

$$-1 = y = 1 + 2y = 1 + 2(1 + 2y) = 1 + 2 + 4y = \dots$$

co rozwija się w szereg

$$-1 = 1 + 2 + 4 + 8 + \dots$$

Chociaż w przypadku obu równań wykonujemy takie same formalne przekształcenia, w pierwszym przypadku otrzymujemy prawdziwe stwierdzenie dotyczące szeregów nieskończonych, a w drugim nie. Podobnie w przypadku wyników unifikacji, wnioskowanie oparte na dowolnie zbudowanym wyrażeniu może prowadzić do błędów.

Depends-on? to predykat, który sprawdza, czy wyrażenie zaproponowane jako wartość zmiennej zależy od tejże zmiennej. Musi być to wykonywane w odniesieniu do bieżącej ramki, gdyż wyrażenie może zawierać wystąpienia zmiennych, w miejscu których podstawiane są już wartości, które zależą od badanej zmiennej. Struktura depends-on? odzwierciedla proste rekurencyjne przechodzenie drzewa, podczas którego podstawiamy w miejscu zmiennych ich wartości, kiedy tylko jest to potrzebne.

```
(define (depends-on? exp var frame)
  (define (tree-walk e)
    (cond ((var? e)
           (if (equal? var e)
               true
               (let ((b (binding-in-frame e frame)))
                 (if b
                     (tree-walk (binding-value b))
                     false))))
          ((pair? e)
           (or (tree-walk (car e))
               (tree-walk (cdr e))))
          (else false)))
  (tree-walk exp))
```

#### 4.4.4.5. Utrzymywanie bazy danych

Gdy projektujemy język programowania w logice, ważną kwestią jest zadbanie o to, aby w trakcie dopasowywania wzorca było badanych jak najmniej nieistotnych pozycji w bazie danych. W naszym systemie oprócz tego, że pamiętamy wszystkie asercje w postaci jednego wielkiego strumienia, pamiętamy w osobnych strumieniach wszystkie te asercje, których cary są symbolami stałych\*, a strumienie te tworzą tablicę indeksowaną tymi symbolami. Pobierając asercję, która może pasować do wzorca, sprawdzamy najpierw, czy car wzorca nie jest symbolem stałej. Jeśli jest on symbolem stałej, to przekazujemy (do procedury dopasowywania wzorców) wszystkie zapamiętane asercje mające taki sam car. Jeśli car wzorca nie jest symbolem stałej, to przekazujemy wszystkie pamiętane asercje. Sprytniejsze metody mogłyby korzystać również z informacji zawartych w ramce lub starać się także optymalizować przypadek, gdy car wzorca nie jest symbolem stałej. Unikamy wbudowywania kryteriów indeksowania (opartych na carach wzorców, w przypadku gdy jest to symbol stałej) w program; zamiast tego składamy to na barki predykatów i selektorów realizujących te kryteria.

---

\* To znaczy, nie są zmiennymi. Mogą to natomiast być ustalone nazwy relacji (przyp. tłum.).

---

```

(define THE-ASSERTIONS the-empty-stream)

(define (fetch-assertions pattern frame)
  (if (use-index? pattern)
      (get-indexed-assertions pattern)
      (get-all-assertions)))

(define (get-all-assertions) THE-ASSERTIONS)

(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))

```

Procedura `get-stream` wyszukuje strumień w tablicy i przekazuje pusty strumień, jeśli w tablicy nie ma szukanego strumienia.

```

(define (get-stream key1 key2)
  (let ((s (get key1 key2)))
    (if s s the-empty-stream)))

```

Reguły są przechowywane podobnie, według carów nagłówków. Jednak nagłówki reguł są dowolnymi wzorcami, różnią się więc od asercji tym, że mogą zawierać zmienne. Wzorzec, którego car jest symbolem stałej, może być dopasowany do reguł, których nagłówki zaczynają się od zmiennej lub też mają taki sam car. Tak więc, pobierając reguły mogące pasować do wzorca, którego car jest symbolem stałej, bierzemy wszystkie reguły, których nagłówki rozpoczynają się od zmiennej, oraz takie, których nagłówki mają taki sam car co wzorzec. W tym celu wszystkie reguły, których nagłówki zaczynają się od zmiennej, pamiętamy w osobnym strumieniu, w elemencie tablicy indeksowanym symbolem `?`.

```

(define THE-RULES the-empty-stream)

(define (fetch-rules pattern frame)
  (if (use-index? pattern)
      (get-indexed-rules pattern)
      (get-all-rules)))

(define (get-all-rules) THE-RULES)

(define (get-indexed-rules pattern)
  (stream-append
    (get-stream (index-key-of pattern) 'rule-stream)
    (get-stream '?' 'rule-stream)))

```

Procedura `add-rule-or-assertion!` jest używana przez `query-driver-loop` do wprowadzania do bazy danych asercji i reguł. Każda pozycja, jeśli trzeba, jest pamiętana w indeksie oraz w strumieniu wszystkich asercji lub reguł w bazie danych.

```

(define (add-rule-or-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-assertion! assertion)))

(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS
          (cons-stream assertion old-assertions)))
  'ok))

(define (add-rule! rule)
  (store-rule-in-index rule)
  (let ((old-rules THE-RULES))
    (set! THE-RULES (cons-stream rule old-rules)))
  'ok))

```

Chcąc faktycznie zapamiętać asercję bądź regułę, sprawdzamy, czy może ona być indeksowana. Jeśli tak, to zapamiętujemy ją w odpowiednim strumieniu.

```

(define (store-assertion-in-index assertion)
  (if (indexable? assertion)
      (let ((key (index-key-of assertion)))
        (let ((current-assertion-stream
              (get-stream key 'assertion-stream)))
          (put key
                'assertion-stream
                (cons-stream assertion
                              current-assertion-stream))))))

(define (store-rule-in-index rule)
  (let ((pattern (conclusion rule)))
    (if (indexable? pattern)
        (let ((key (index-key-of pattern)))
          (let ((current-rule-stream
                (get-stream key 'rule-stream)))
            (put key
                  'rule-stream
                  (cons-stream rule
                                current-rule-stream)))))))

```

Następujące procedury definiują sposób użycia indeksu. Wzorzec (asercja lub nagłówek reguły) jest zapamiętywany w tablicy, jeśli zaczyna się od zmiennej lub symbolu stałej.

```
(define (indexable? pat)
  (or (constant-symbol? (car pat))
      (var? (car pat))))
```

Klucz, pod którym wzorzec jest zapamiętywany w tablicy, jest albo symbolem ? (jeśli zaczyna się od zmiennej), albo symbolem stałej, od którego się zaczyna.

```
(define (index-key-of pat)
  (let ((key (car pat)))
    (if (var? key) '? key)))
```

Jeśli wzorzec zaczyna się od symbolu stałej, to do pobrania pozycji, które mogą pasować do wzorca, możemy użyć indeksu.

```
(define (use-index? pat)
  (constant-symbol? (car pat)))
```

### Ćwiczenie 4.70

W jakim celu w procedurach `add-assertion!` i `add-rule!` używa się wyrażeń `let?` Co zlego jest w następującej implementacji `add-assertion!`? Wskazówka: Przypomnij sobie definicję nieskończonego strumienia jedynek z punktu 3.5.2: (`define ones (cons-stream 1 ones)`).

```
(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (set! THE-ASSERTIONS
    (cons-stream assertion THE-ASSERTIONS)))
  'ok)
```

#### 4.4.4.6. Operacje na strumieniach

System zapytań korzysta z kilku operacji na strumieniach, których nie przedstawiliśmy w rozdziale 3.

Procedury `stream-append-delayed` i `interleave-delayed` są takie jak `stream-append` i `interleave` (z punktu 3.5.3) z wyjątkiem tego, że ich argumenty są odroczone (jak w przypadku procedury `integral` z punktu 3.5.4). W niektórych przypadkach zapobiega to zapętlom (zob. ćwiczenie 4.71).

```
(define (stream-append-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
        (stream-car s1)
        (stream-append-delayed (stream-cdr s1) delayed-s2))))
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
```

```
(cons-stream
  (stream-car s1)
  (interleave-delayed (force delayed-s2)
    (delay (stream-cdr s1)))))
```

Procedura **stream-flatmap**, która jest używana w całym ewaluatorze za-pytań do przekształcania strumienia ramek zgodnie z daną procedurą oraz łączenia wynikowych strumieni ramek, jest strumieniowym odpowiednikiem procedury **flatmap** dla zwykłych list, wprowadzonej w punkcie 2.2.3. W odróżnieniu jednak od zwykłego **flatmap** tutaj kumulujemy strumienie, prze-platając je, a nie po prostu doklejając jeden na końcu drugiego (zob. ćwicze-nia 4.72 i 4.73).

```
(define (stream-flatmap proc s)
  (flatten-stream (stream-map proc s)))

(define (flatten-stream stream)
  (if (stream-null? stream)
    the-empty-stream
    (interleave-delayed
      (stream-car stream)
      (delay (flatten-stream (stream-cdr stream))))))
```

Ewaluator generuje jednoelementowe strumienie za pomocą następującej prostej procedury:

```
(define (singleton-stream x)
  (cons-stream x the-empty-stream))
```

#### 4.4.4.7. Procedury składni zapytań

Procedury **type** i **contents**, używane przez **qeval** (podpunkt 4.4.4.2), okre-slają, że forma specjalna jest rozpoznawana na podstawie symbolu stanowią-cego jej **car**. Są one takie same jak procedury **type-tag** i **contents** z punk-tu 2.4.2, z wyjątkiem komunikatu o błędzie.

```
(define (type exp)
  (if (pair? exp)
    (car exp)
    (error "Nieznany RODZAJ wyrażenia" exp)))

(define (contents exp)
  (if (pair? exp)
    (cdr exp)
    (error "Nieznana TREŚĆ wyrażenia" exp)))
```

Następujące procedury, używane przez query-driver-loop (w podpunkcie 4.4.4.1), określają, że reguły i asercje są dodawane do bazy danych za pomocą wyrażeń postaci (`assert! (reguła lub asercja)`):

```
(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))
```

```
(define (add-assertion-body exp)
  (car (contents exp)))
```

Oto definicje składni form specjalnych `and`, `or`, `not` i `lisp-value` (podpunkt 4.4.4.2):

```
(define (empty-conjunction? exps) (null? exps))
(define (first-conjunct exps) (car exps))
(define (rest-conjuncts exps) (cdr exps))
```

```
(define (empty-disjunction? exps) (null? exps))
(define (first-disjunct exps) (car exps))
(define (rest-disjuncts exps) (cdr exps))
```

```
(define (negated-query exps) (car exps))
```

```
(define (predicate exps) (car exps))
(define (args exps) (cdr exps))
```

Następujące trzy procedury definiują składnię reguł:

```
(define (rule? statement)
  (tagged-list? statement 'rule))
```

```
(define (conclusion rule) (cadr rule))
```

```
(define (rule-body rule)
  (if (null? (cddr rule))
    '(always-true)
    (caddr rule)))
```

Procedura query-driver-loop (podpunkt 4.4.4.1) wywołuje query-syntax-process w celu przekształcenia występujących w wyrażeniu zmiennych wzorca, które mają postać `?symbol`, na wewnętrzny format (`(? symbol)`). Oznacza to, że taki wzorzec jak (`job ?x ?y`) jest faktycznie reprezentowany wewnętrznie przez system jako (`job (? x) (? y)`). Zwiększa to efektywność przetwarzania zapytań, gdyż dzięki temu system może sprawdzić, czy wyrażenie jest zmienną wzorca, sprawdzając, czy `car` wyrażenia jest symbo-

lem ?, a nie musi dzielić symbolu na znaki. To przekształcenie składni jest wykonywane przez następującą procedurę<sup>80,\*</sup>:

```
(define (query-syntax-process exp)
  (map-over-symbols expand-question-mark exp))

(define (map-over-symbols proc exp)
  (cond ((pair? exp)
         (cons (map-over-symbols proc (car exp))
               (map-over-symbols proc (cdr exp))))
         ((symbol? exp) (proc exp))
         (else exp)))

(define (expand-question-mark symbol)
  (let ((chars (symbol->string symbol)))
    (if (string=? (substring chars 0 1) "?")
        (list '? 
              (string->symbol
                (substring chars 1 (string-length chars))))
        symbol)))
```

Gdy zmienne zostaną już przekształcone w ten sposób, wówczas są one listami zaczynającymi się od symbolu ?, a symbole stałych (które muszą być odróżniane ze względu na indeksowanie bazy danych, podpunkt 4.4.4.5) są po prostu symbolami.

```
(define (var? exp)
  (tagged-list? exp '?))

(define (constant-symbol? exp) (symbol? exp))
```

Niepowtarzalne zmienne są konstruowane w trakcie stosowania reguł (w podpunkcie 4.4.4.4) za pomocą następujących procedur. Niepowtarzalny

<sup>80</sup> Większość systemów lispowych umożliwia użytkownikowi modyfikowanie zwykłej procedury `read`, aby wykonywała ona takie przekształcenia przez jednoznakowe makrodefinicje wczytywania. Cytowane wyrażenia są właśnie obsługiwane w ten sposób: procedura wczytująca automatycznie tłumaczy '`wyrażenie`' na `(quote <wyrażenie>)`, zanim jeszcze zobaczy je evaluator. Moglibyśmy tak zrobić, aby w ten sam sposób `?<wyrażenie>` było zamieniane na `(? <wyrażenie>)`, jednak dla większej przejrzystości dołączylismy tutaj jawnie procedurę przekształcającą.

Expand-question-mark i contract-question-mark korzystają z kilku procedur zawierających człon `string` w swoich nazwach. Są to operacje pierwotne języka Scheme.

\* `Symbol->string` przekształca nazwę danego symbolu w napis. `String=?` porównuje dwa dane napisy. Wynikiem `substring` jest wskazany fragment danego napisu; fragment ten jest wskazywany przez określenie, ile początkowych znaków należy pominąć i na którym znaku danego napisu kończy się fragment. Wynikiem `string->symbol` jest symbol o nazwie zadanej w postaci napisu. `String-length` oblicza długość danego napisu (przyp. tłum.).

identyfikator zastosowania reguły jest liczbą, która jest zwiększana przy każdym zastosowaniu reguły.

```
(define rule-counter 0)

(define (new-rule-application-id)
  (set! rule-counter (+ 1 rule-counter))
  rule-counter)

(define (make-new-variable var rule-application-id)
  (cons '? (cons rule-application-id (cdr var))))
```

Gdy query-driver-loop stosuje podstawienie do zapytania, aby wypisać odpowiedź, wówczas przekształca ona każdą zmienną wolną z powrotem na postać właściwą do wypisania, wykorzystując przy tym\*

```
(define (contract-question-mark variable)
  (string->symbol
    (string-append "?"
      (if (number? (cadr variable))
        (string-append (symbol->string (caddr variable))
          "_"
          (number->string (cadr variable)))
        (symbol->string (cadr variable)))))))
```

#### 4.4.4.8. Ramki i wiązania

Ramki są reprezentowane jako listy wiązań, które mają postać par zmienna-wartość:

```
(define (make-binding variable value)
  (cons variable value))

(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (binding-in-frame variable frame)
  (assoc variable frame))

(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

---

\* String-append łączy podane napisy w jeden. Number->string przekształca liczbę w jej zapis (przyp. tłum.).

### Ćwiczenie 4.71

Ludwik Myślicielak zastanawia się, dlaczego implementacje procedur `simple-query` i `disjoin` (podpunkt 4.4.4.2) używają jawnie operacji `delay`, a nie są zdefiniowane w następujący sposób:

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append (find-assertions query-pattern frame)
                   (apply-rules query-pattern frame)))
   frame-stream))

(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave
       (qevel (first-disjunct disjuncts) frame-stream)
       (disjoin (rest-disjuncts disjuncts) frame-stream))))
```

Czy możesz podać przykłady zapytań, dla których te prostsze definicje prowadziłyby do niepożądanego zachowania?

### Ćwiczenie 4.72

Dlaczego `disjoin` i `stream-flatmap` przeplatają strumienie, a nie po prostu łączą je jeden za drugim? Podaj przykłady ilustrujące, dlaczego przeplatanie daje lepsze wyniki. (Wskazówka: Dlaczego używaliśmy `interleave` w punkcie 3.5.3?)

### Ćwiczenie 4.73

Dlaczego `flatten-stream` używa jawnie `delay`? Co byłoby źle, gdybyśmy zdefiniowali tę procedurę w następujący sposób:

```
(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave
       (stream-car stream)
       (flatten-stream (stream-cdr stream)))))
```

### Ćwiczenie 4.74

Liz P. Haker zaproponowała zastosowanie prostszej wersji `stream-flatmap` w `negate`, `lisp-value` i `find-assertions`. Zauważa ona, że w tych przypadkach wynikiem procedury przekształcającej ramki tworzące strumień jest zawsze albo pusty strumień, albo strumień jednoelementowy, a więc żadne przeplatanie nie jest konieczne przy łączeniu tych strumieni.

(a) Uzupełnij brakujące wyrażenia w programie Liz.

```
(define (simple-stream-flatmap proc s)
  (simple-flatten (stream-map proc s)))
```

```
(define (simple-flatten stream)
  (stream-map (??)
              (stream-filter (??) stream)))
```

(b) Czy zachowanie systemu zapytań się zmieni, jeśli wprowadzimy te zmiany?

### Ćwiczenie 4.75

Zaimplementuj w języku zapytań nową formę specjalną o nazwie `unique`. Dopasowanie `unique` powinno się udawać, jeśli istnieje dokładnie jedna pozycja w bazie danych spełniająca podane zapytanie. Na przykład

```
(unique (job ?x (computer wizard)))
```

powinno spowodować wypisanie jednoelementowego strumienia

```
(unique (job (Bajerbit Ben) (computer wizard)))
```

gdyż Ben jest jedynym magikiem komputerowym; z kolei

```
(unique (job ?x (computer programmer)))
```

powinno spowodować wypisanie pustego strumienia, gdyż istnieje więcej niż jeden programista komputerowy. Ponadto

```
(and (job ?x ?j) (unique (job ?anyone ?j)))
```

powinno spowodować wypisanie wszystkich stanowisk, które są obsadzone przez tylko jedną osobę, oraz osób zajmujących te stanowiska.

Zaimplementowanie `unique` składa się z dwóch części. Pierwsza z nich polega na napisaniu procedury przetwarzającej tę formę specjalną, a druga polega na sprawieniu, aby `qeval` kierowała do tej procedury wyrażenia `unique`. Druga część jest trywialna, gdyż `qeval` rozdziela wyrażenia w sposób sterowany danymi. Jeśli nazwiesz swoją procedurę `uniquely-asserted`, to wystarczy, że wykonasz

```
(put 'unique 'qeval uniquely-asserted)
```

a `qeval` będzie kierować do tej procedury wszystkie zapytania, których rodzaj (`type`), czyli `car`, jest symbolem `unique`.

Prawdziwy problem polega na napisaniu procedury `uniquely-asserted`. Jej argumentami powinny być treść (`contents`), czyli `cdr`, zapytania `unique` oraz strumień ramek. Dla każdej ramki z tego strumienia powinna ona wyznaczać, za pomocą `qeval`, strumień wszystkich rozszerzeń danej ramki, które spełniają dane zapytanie. Każdy strumień, który nie zawiera dokładnie jednego elementu, powinien być usuwany. Pozostałe strumienie powinny być przekazywane z powrotem, aby można je było połączyć w jeden wielki strumień będący wynikiem zapytania `unique`. Jest to podobne do implementacji formy specjalnej `not`.

Przetestuj swoją implementację, formułując zapytanie, które wypisuje wszystkie osoby mające dokładnie jednego podwładnego.

### Ćwiczenie 4.76

Nasza implementacja `and` w postaci złożenia kolejnych zapytań (rys. 4.5) jest elegancka, ale nieefektywna. Dzieje się tak, gdyż przetwarzając drugie zapytanie wchodzące w skład `and`, musimy dla każdej ramki wygenerowanej przez pierwsze zapytanie przejrzeć bazę danych. Gdyby baza danych zawierała  $N$  elementów, a liczba ramek będących wynikiem typowego zapytania była proporcjonalna do  $N$  (powiedzmy równa  $N/k$ ), to przeszukiwanie bazy danych dla każdej ramki będącej wynikiem pierwszego zapytania wymagałoby  $N^2/k$  wywołań procedury dopasowującej wzorce. Inne podejście mogłoby polegać na przetwarzaniu obu klauzul `and` osobno, a następnie znajdowaniu wszystkich takich par wynikowych ramek, które są ze sobą zgodne. Jeśli każde z zapytań daje w wyniku  $N/k$  ramek, to oznacza to wykonanie  $N^2/k^2$  sprawdzeń zgodności — czyli  $k$  razy mniej niż liczba dopasowań wzorców wymagana przy aktualnie stosowanej metodzie.

Obmyśl implementację `and`, która stosuje opisaną strategię. Musisz zaimplementować procedurę, która otrzymuje dwie ramki, sprawdza, czy odpowiadające im podstawnienia są zgodne, a jeśli tak, to tworzy ramkę, która stanowi scalenie tych dwóch podstawnień. Operacja ta przypomina unifikację.

### Ćwiczenie 4.77

W punkcie 4.4.3 widzieliśmy, że `not` i `lisp-value` mogą powodować, iż język zapytań daje „złe” odpowiedzi, jeśli te operacje filtrujące są zastosowane do ramek, w których niektóre zmienne są wolne. Obmyśl sposób usunięcia tego mankamentu. Jeden z pomysłów polega na „odraczaniu” filtrowania przez dołączenie do ramki „obietnicy” przepuszczenia jej przez filtr, gdy wystarczająco dużo zmiennych będzie związkowych, aby można było to wykonać. Moglibyśmy czekać z filtrowaniem do momentu, gdy wszystkie inne operacje zostaną wykonane. Jednak ze względu na efektywność chciałibyśmy, żeby filtrowanie zachodziło jak najwcześniej, aby zmniejszyć liczbę generowanych ramek pośrednich.

### Ćwiczenie 4.78

Zaprojektuj ponownie język zapytań jako program niedeterministyczny, tak aby można było go zaimplementować za pomocą evaluatora z podrozdziału 4.3 zamiast przetwarzania strumieni. Przy takim podejściu każde zapytanie będzie dawać w wyniku pojedynczą odpowiedź (zamiast strumienia wszystkich odpowiedzi), a użytkownik może wprowadzać `try-again`, jeśli chce zobaczyć więcej odpowiedzi. Powinieneś przy tym odkryć, że wiele mechanizmów, które zbudowaliśmy w niniejszym podrozdziale, jest realizowanych w ramach niedeterministycznego przeszukiwania i nawracania. Prawdopodobnie jednak dostrzeżesz też, że istnieją subtelne różnice w działaniu Twojego nowego języka zapytań i tego zaimplementowanego tutaj. Czy potrafisz wskazać przykłady ilustrujące takie różnice?

### Ćwiczenie 4.79

Gdy w podrozdziale 4.1 implementowaliśmy evaluator Lispu, widzieliśmy, jak można wykorzystać środowiska lokalne w celu uniknięcia konfliktów nazw między parametrami procedur.

Na przykład w trakcie obliczania

```
(define (square x)
  (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
```

nie ma możliwości pomylenia ze sobą `x` z procedury `square` oraz `x` z `sum-of-squares`, gdyż treść każdej z tych procedur obliczamy w specjalnie utworzonym środowisku, takim, które zawiera wiązania zmiennych lokalnych. W systemie zapytań użyliśmy innej strategii w celu uniknięcia konfliktów nazw przy stosowaniu reguł. Za każdym razem, gdy stosujemy regułę, przemianowujemy zmienne, nadając im niepowtarzalne nazwy. W przypadku evaluatora Lispu analogiczna strategia polegałaby na pozbyciu się środowisk lokalnych i po prostu przemianowywaniu zmiennych w treści procedury za każdym razem, gdy stosujemy procedurę.

Zaimplementuj metodę stosowania reguł w języku zapytań, która korzysta ze środowisk zamiast z przemianowywania zmiennych. Zobacz, czy można użyć struktury środowisk do utworzenia w języku zapytań konstruktorów służących do radzenia sobie z dużymi systemami, takich jak odpowiedniki procedur o strukturze blokowej dla reguł. Czy potrafisz odnieść cokolwiek z tego do problemu wnioskowania w kontekście (np. „Gdybym założył, że  $P$  jest prawdziwe, to mógłbym pokazać, że  $A$  i  $B$  też są prawdziwe.”) jako metody rozwiązywania problemów? (Jest to otwarte zagadnienie. Dobra odpowiedź jest prawdopodobnie warta doktoratu).

# 5

## Obliczenia z użyciem maszyn rejestrowych

Moim celem jest wykazanie, że niebiańska maszyneria nie jest rodzajem boskiej istoty, lecz pewnego rodzaju mechanizmem zegarowym (a kto wierzy, że zegar ma duszę, ten przypisuje mu chwałę jego twórcy), tak dalece, że prawie wszystkie różnorakie ruchy są powodowane najprostszą i materialną siłą, tak jak wszystkie ruchy zegara są powodowane jednym ciężarkiem.

Johannes Kepler (list do Herwarta von Hohenburga, 1605 r.)

Na początku niniejszej książki badaliśmy procesy i przedstawialiśmy je w postaci procedur zapisanych w Lispie. Do wyjaśnienia znaczenia tych procedur użyliśmy szeregu modeli obliczeń: modelu podstawieniowego z rozdziału 1, modelu środowiskowego z rozdziału 2 i ewaluatora metacyklicznego z rozdziału 4. Zwłaszcza zbadanie ewaluatora metacyklicznego pozwoliło wyjaśnić większość tajemnic dotyczących sposobu interpretacji języków lispopodobnych. Jednak nawet opis działania ewaluatora metacyklicznego pozostawia niektóre istotne kwestie bez odpowiedzi, gdyż nie wyjaśnia mechanizmów sterowania w systemie lispowym. Na przykład z konstrukcji ewaluatora nie wynika, w jaki sposób w trakcie obliczania wyrażeń wyniki podwyrażeń są przekazywane do miejsca ich użycia; nie wynika z niej również, w jaki sposób dla niektórych procedur rekurencyjnych generowane są procesy iteracyjne (tzn. obliczane w stałej pamięci), podczas gdy inne procedury rekurencyjne generują procesy rekurencyjne. Kwestie te pozostają niewyjaśnione, gdyż ewaluator metacykliczny sam jest programem w Lispie, dzięki czemu dziedziczy strukturę sterowania z działającego pod spodem systemu lispowego. Chcąc przedstawić pełniejszy opis struktury sterowania ewaluatora Lispu, musimy posługiwać się pojęciami z niższego poziomu niż sam Lisp.

W niniejszym rozdziale opiszemy działanie procesów za pomocą kolejnych operacji tradycyjnego komputera. Taki komputer, bądź *maszyna rejestrowa* (ang. *register machine*), sekwencyjnie wykonuje *instrukcje* (rozkazy) operujące na zawartości ustalonego zestawu elementów pamięciowych nazywanych *rejestrami*. Typowa instrukcja maszyny rejestrowej powoduje wykonanie pewnej operacji pierwotnej na zawartościach pewnych rejestrów i umieszczenie jej wyniku w innym rejestrze. Nasze opisy procesów wykonywanych przez

maszyny rejestrowe będą w dużym stopniu wyglądać jak programy w „języku maszynowym” dla tradycyjnych komputerów. Jednak zamiast koncentrować się na języku maszynowym jakiegokolwiek konkretnego komputera, zbadamy szereg procedur lispowych i dla każdej z nich zaprojektujemy określoną maszynę rejestrową wykonującą daną procedurę. Tak więc podejdziemy do tego zadania raczej od strony architektury sprzętu, a nie programowania w języku maszynowym. Projektując maszyny rejestrowe, opracujemy mechanizmy implementujące ważne konstrukcje programowania, takie jak rekursja. Przedstawimy również język służący do opisywania projektów maszyn rejestrowych. W podrozdziale 5.2 zaimplementujemy program w Lispie, który na podstawie takich opisów może symulować zaprojektowane przez nas maszyny.

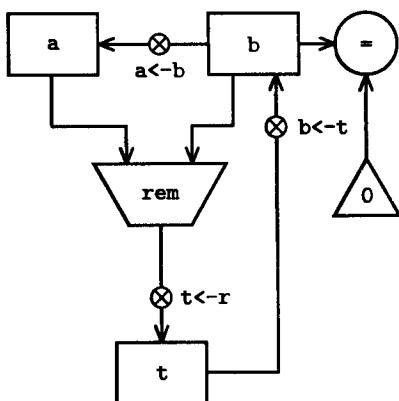
Większość operacji pierwotnych stosowanych w maszynach rejestrowych jest bardzo prosta. Operacja pierwotna może na przykład dodawać liczby pobrane z dwóch rejestrów i zapamiętywać uzyskaną sumę w trzecim rejestrze. Operacja taka może być wykonywana sprzętowo w łatwy do opisania sposób. Jednakże chcąc operować na strukturach listowych, będziemy również używać operacji na pamięci: `car`, `cdr` i `cons`, które wymagają wyszukanego mechanizmu przydzielania pamięci. W podrozdziale 5.3 zajmiemy się ich implementacją przy wykorzystaniu bardziej elementarnych operacji.

W podrozdziale 5.4, po tym jak zdobędziemy doświadczenie w formułowaniu prostych procedur w postaci maszyn rejestrowych, zaprojektujemy maszynę wykonującą algorytm opisany przez evaluator metacykliczny z podrozdziału 4.1. Uzupełni to nasze zrozumienie procesu interpretacji wyrażeń w języku Scheme, dzięki jawnemu określeniu modelu mechanizmu sterowania w evaluatorze. W podrozdziale 5.5 zbadamy prosty kompilator, który tłumaczy programy w języku Scheme na ciągi instrukcji, które mogą być wykonywane bezpośrednio za pomocą rejestrów i operacji maszyny rejestrowej evaluatora.

## 5.1. Projektowanie maszyn rejestrowych

Chcąc zaprojektować maszynę rejestrową, musimy zaprojektować jej *ścieżki danych* (ang. *data paths*; rejesty i operacje) oraz *sterownik* (ang. *controller*), który wykonuje te operacje w odpowiedniej kolejności. Aby zilustrować projekt prostej maszyny rejestrowej, zbadajmy algorytm Euklidesa używany do obliczania największego wspólnego dzielnika (NWD) dwóch liczb całkowitych. Jak widzieliśmy to w punkcie 1.2.5, algorytm Euklidesa może być wykonywany przez proces iteracyjny, określony następującą procedurą:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```



Rys. 5.1. Ścieżki danych maszyny obliczającej NWD

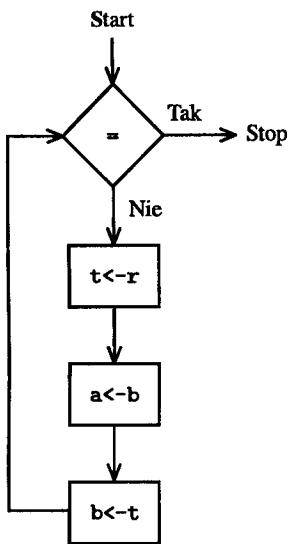
Maszyna wykonująca ten algorytm musi pamiętać dwie liczby  $a$  i  $b$ . Założymy więc, że liczby te są pamiętane w dwóch rejestrach o takich samych nazwach. Podstawowe operacje, które muszą być wykonywane, to sprawdzenie, czy zawartość rejestru  $b$  jest równa zeru, i obliczanie reszty z dzielenia zawartości rejestru  $a$  przez zawartość rejestru  $b$ . Obliczanie reszty z dzielenia to złożony proces, ale założymy na razie, że mamy pierwotne urządzenie, które oblicza reszty. W każdym cyklu algorytmu NWD zawartość rejestru  $a$  musi być zastąpiona przez zawartość rejestru  $b$ , a zawartość rejestru  $b$  musi być zastąpiona przez resztę z dzielenia starej zawartości  $a$  przez starą zawartość  $b$ . Wygodnie by było, gdyby te zastąpienia mogły być wykonywane równocześnie, jednak w naszym modelu maszyn rejestrowych zakładamy, że w każdym kroku można przypisać nową wartość tylko jednemu rejestrowi. Do realizacji przypisań nasza maszyna będzie używać trzeciego, „tymczasowego” rejestrów, który będziemy oznaczać przez  $t$ . (Najpierw reszta z dzielenia będzie umieszczana w  $t$ , następnie zawartość  $b$  będzie umieszczana w  $a$  i na koniec reszta z dzielenia zapamiętana w  $t$  będzie umieszczana w  $b$ ).

Rejestry i operacje wymagane do skonstruowania takiej maszyny możemy zilustrować za pomocą diagramu ścieżek danych, pokazanego na rys. 5.1. Na diagramie tym rejesty ( $a$ ,  $b$  i  $t$ ) są przedstawione jako prostokąty. Z kolei każdy sposób przypisania wartości do rejestrów jest przedstawiony jako strzałka z symbolem  $\otimes$ , prowadząca od wartości przypisywanej do miejsca docelowego, w którym wartość ta jest zapamiętywana. O symbolu  $\otimes$  możemy myśleć jako o przycisku, który w momencie naciśnięcia pozwala „przepłynąć” wartości z początku strzałki do wskazywanego przez nią docelowego rejestrów. Obok przycisków są umieszczone nazwy, których będziemy używać do ich oznaczenia. Nazwy te mogą być dowolne i możemy je tak wybierać, aby łatwo można było zapamiętać ich znaczenie (np.  $a <- b$  oznacza naciśnięcie przycisku, które

powoduje przypisanie do rejestru a zawartości rejestru b). Dane umieszczone w rejestrze mogą pochodzić z innego rejestru (jak w przypadku przypisania  $a \leftarrow b$ ), mogą być wynikiem operacji (jak w przypadku przypisania  $t \leftarrow r$ ) lub może to być stała (wartość wbudowana, której nie można zmienić, reprezentowana na diagramach ścieżek danych przez trójkąt zawierający stałą).

Operacje, których argumentami są stałe i/lub zawartości rejestrów, są reprezentowane na diagramach ścieżek danych za pomocą trapezów zawierających nazwy operacji. Na rysunku 5.1 widać na przykład trapez zawierający nazwę `rem`, który reprezentuje operację obliczania reszty z dzielenia zawartości podłączonych do niego rejestrów a i b. Strzałki bez przycisków prowadzą od rejestrów i/lub stałych wejściowych do trapezu, a strzałki z przyciskami łączą wyniki operacji z rejestrami. Sprawdzanie warunku jest reprezentowane za pomocą kółka zawierającego nazwę warunku. Nasza maszyna obliczająca NWD zawiera na przykład operację sprawdzającą, czy zawartość rejestru b jest równa零. Strzałki prowadzą również od argumentów warunku do warunku, natomiast żadne strzałki nie wychodzą z warunków; wartość warunku jest używana przez sterownik, a nie w ścieżkach danych. Ogólnie mówiąc, na diagramie ścieżek danych są pokazane rejesty i operacje wymagane do skonstruowania maszyny, a także sposób ich połączenia. Jeśli wyobrażymy sobie, że strzałki to przewody, a przyciski  $\otimes$  to przełączniki, to diagram ścieżek danych będzie przypominał schemat budowy maszyny, którą można skonstruować z części elektronicznych.

Aby ścieżki danych faktycznie umożliwiały obliczenie NWD, przyciski muszą być naciskane w odpowiedniej kolejności. Opiszemy tę kolejność, wykorzystując do tego diagram sterownika przedstawiony na rys. 5.2. Elementy diagramu sterownika wskazują, jak należy posługiwać się składowymi ścieżkami danych. Prostokąty na diagramie oznaczają przyciski ścieżek danych, które należy nacisnąć, a strzałki opisują kolejno wykonywane kroki. Romby na diagramie reprezentują podejmowane wybory. W zależności od wartości warunku z diagramu ścieżek danych (podanego wewnątrz rombu) jako kolejny zostanie wykonany krok wskazywany przez jedną z dwóch strzałek wychodzących z rombu. Działanie sterownika możemy sobie przedstawić za pomocą następującej „mechanicznej” analogii: Wyobraźmy sobie, że diagram reprezentuje labirynt, przez który toczy się kulka. Gdy kulka wpada do prostokątnego pudełka, naciska przycisk, którego nazwa znajduje się na tym pudełku. Gdy zaś wpada do pudełka wyboru (takiego jak warunek sprawdzający, czy  $b = 0$ ), wtedy wylatuje w kierunku określonym przez wartość warunku. Ścieżki danych razem ze sterownikiem w pełni opisują maszynę obliczającą NWD. Po umieszczeniu liczb w rejestrach a i b sterownik rozpoczyna działanie (czyli początkowo umieszcza my kulkę) w miejscu oznaczonym przez Start. Gdy sterownik osiągnie Stop, wartość NWD znajdzie się w rejestrze a.



Rys. 5.2. Sterownik maszyny obliczającej NWD

### Ćwiczenie 5.1

Zaprojektuj maszynę rejestrową obliczającą silnię za pomocą algorytmu iteracyjnego określonego przez poniższą procedurę. Narysuj diagramy ścieżek danych i sterownika dla tej maszyny.

```

(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
  
```

#### 5.1.1. Język opisu maszyn rejestrowych

Diagramy ścieżek danych i sterowników są odpowiednie do reprezentowania prostych maszyn, takich jak ta obliczająca NWD, nie nadają się jednak do reprezentowania dużych maszyn, takich jak interpretera Lispu. Aby móc radzić sobie ze złożonymi maszynami, stworzymy język przedstawiający, w postaci tekstuowej, wszystkie informacje zawarte na diagramach ścieżek danych i sterowników. Zaczniemy od notacji, która dokładnie odzwierciedla diagramy.

Ścieżki danych maszyny definiujemy, opisując rejestr (ang. *registers*) i operacje (ang. *operations*). Aby opisać rejestr, podajemy jego nazwę i określamy przyciski (ang. *buttons*) sterujące przypisaniem do tego rejestru. Dla każdej

go z przycisków podajemy jego nazwę i określamy źródło danych, czyli skąd pochodzą dane przypisywane do rejestru w przypadku naciśnięcia przycisku. Źródłem (ang. *source*) może być rejestr, stała (ang. *constant*) lub operacja. Aby opisać operację, podajemy jej nazwę i określamy jej dane wejściowe (ang. *inputs*) (czyli argumenty, którymi mogą być rejesty lub stałe).

Sterownik maszyny (ang. *controller*) definiujemy za pomocą ciągu *instrukcji* (ang. *instructions*) oraz *etykiet* (ang. *labels*) określających kolejne *punkty wejścia* (ang. *entry points*). Instrukcją może być:

- Nazwa przycisku ścieżki danych, którego naciśnięcie powoduje przypisanie wartości do rejestru. (Odpowiada ona prostokątowi na diagramie sterownika).
- Instrukcja **test** sprawdzająca określony warunek.
- Skok warunkowy (instrukcja **branch**) prowadzący, w zależności od uprzednio badanego warunku, do miejsca wskazywanego przez etykietę. (Warunek wraz ze skokiem warunkowym odpowiadają rombowi na diagramie sterownika). Jeśli warunek nie jest spełniony, sterownik powinien jako następną wykonać kolejną instrukcję. W przeciwnym razie sterownik powinien jako następną wykonać pierwszą instrukcję występującą po etykiecie.
- Skok bezwarunkowy (instrukcja **goto**) określająca etykietę, od której sterownik kontynuuje działanie.

Maszyna zaczyna wykonywanie ciągu instrukcji sterownika od początku i zatrzymuje się, gdy dochodzi do końca ciągu. Z wyjątkiem sytuacji gdy skoki zmieniają przepływ sterowania, instrukcje są wykonywane w takiej kolejności, w jakiej są podane.

Na rysunku 5.3 jest pokazany taki właśnie opis maszyny obliczającej NWD. Przykład ten tylko w niewielkim stopniu ukazuje ogólność takich opisów, gdyż maszyna obliczająca NWD jest bardzo prosta — z każdym rejestrem jest związany tylko jeden przycisk, a każdy przycisk i warunek jest używany w sterowniku tylko raz.

Niestety, bardzo trudno czyta się takie opisy. Chcąc zrozumieć instrukcje sterownika, musimy cały czas odwoływać się do definicji nazw przycisków i nazw operacji, a żeby zrozumieć, do czego służą przyciski, również musimy odwoływać się do definicji nazw operacji. Dlatego też przekształcimy naszą notację, łącząc informacje pochodzące z opisów ścieżek danych i sterownika, tak aby całość była widoczna razem.

Aby uzyskać taką postać opisów, zastąpimy dowolne nazwy przycisków i operacji definicjami ich zachowania. Oznacza to, że zamiast osobno mówić (w sterowniku) „naciśnij przycisk  $t <- r$ ”, (w ścieżkach danych) „przycisk  $t <- r$  przypisuje rejestrowi  $t$  wynik operacji  $r \text{em}$ ” oraz „argumentami operacji  $r \text{em}$

```

(data-paths
  (registers
    ((name a)
      (buttons ((name a<-b) (source (register b))))))
    ((name b)
      (buttons ((name b<-t) (source (register t))))))
    ((name t)
      (buttons ((name t<-r) (source (operation rem))))))

  (operations
    ((name rem)
      (inputs (register a) (register b)))
    ((name =)
      (inputs (register b) (constant 0)))))

(controller
  test-b
    (test =) ; etykieta
    (branch (label gcd-done)) ; warunek
    (t<-r) ; skok warunkowy
    (a<-b) ; naciśnięcie przycisku
    (b<-t) ; naciśnięcie przycisku
    (goto (label test-b)) ; skok bezwarunkowy
  gcd-done) ; etykieta

```

Rys. 5.3. Specyfikacja maszyny obliczającej NWD

są zawartości rejestrów a i b”, powiemy (w sterowniku) „naciśnij przycisk, który przypisuje rejestrowi t wynik operacji rem zastosowanej do rejestrów a i b”. Podobnie, zamiast osobno mówić (w sterowniku) „sprawdź warunek =” i (w ścieżkach danych) „argumentami warunku = jest zawartość rejestrów b i stała 0”, powiemy „sprawdź warunek = dla zawartości rejestrów b i stałej 0”. Pominiemy opis ścieżek danych, pozostawiając jedynie ciąg opisujący sterownik. Tak więc maszyna obliczająca NWD może być opisana w następujący sposób:

```

(controller
  test-b
    (test (op =) (reg b) (const 0))
    (branch (label gcd-done))
    (assign t (op rem) (reg a) (reg b))
    (assign a (reg b))
    (assign b (reg t))
    (goto (label test-b))
  gcd-done)

```

Opisy w takiej postaci czyta się łatwiej niż te przedstawione na rys. 5.3, jednak mają one również wady:

- Są bardziej rozwlekłe w przypadku dużych maszyn, gdyż za każdym razem, gdy w sekwencji instrukcji sterownika odwołujemy się do ścieżek danych, powtarzane są ich pełne opisy. (Nie stanowi to problemu w przykładzie z NWD, gdyż każda operacja i każdy przycisk jest używany tylko raz). Co więcej, powtarzanie opisów ścieżek danych zaciemnia tylko faktyczną strukturę maszyny; w przypadku dużej maszyny nie jest oczywiste, ile zawiera ona rejestrów, operacji oraz przycisków i jak są one ze sobą połączone.
- Ponieważ w definicji maszyny instrukcje sterownika wyglądają jak wyrażenia lispowe, łatwo można zapomnieć, że nie mogą to być dowolne wyrażenia. Mogą one przedstawiać tylko poprawne operacje maszynowe. Na przykład argumentami operacji mogą być bezpośrednio tylko stałe i zawartości rejestrów, a nie wyniki innych operacji.

Pomimo tych wad będziemy w niniejszym rozdziale używać właśnie takiego języka opisu maszyn rejestrowych, gdyż bardziej nas będzie interesować zrozumienie działania sterowników niż poznanie struktury elementów i połączeń tworzących ścieżki danych. Powinniśmy jednak pamiętać, że projekt ścieżek danych odgrywa kluczową rolę w projektowaniu rzeczywistych maszyn.

### Ćwiczenie 5.2

Użyj języka opisu maszyn rejestrowych do opisania maszyny z ćwiczenia 5.1 obliczającej iteracyjnie silnię.

### Akceje

Zmodyfikujmy tak maszynę obliczającą NWD, aby można było wpisywać liczby, których NWD ma być obliczony, i aby wynik był wypisywany na terminalu. Nie będziemy omawiali tego, jak można zbudować maszynę, która potrafi wczytywać i wypisywać informacje, lecz założymy (tak jak to robimy, gdy używamy w języku Scheme operacji `read` i `display`), że wczytanie i wypisywanie są dostępne jako operacje pierwotne<sup>1</sup>.

Read przypomina operacje, których używaliśmy, w tym sensie, że wynikiem tej operacji jest wartość, którą można zapamiętać w rejestrze. Jednak `read` nie pobiera żadnych argumentów z rejestrów; wynik tej operacji zależy od czegoś, co zachodzi poza częścią maszyny, którą projektujemy. Dopuszczamy, żeby operacje naszej maszyny zachowywały się w ten sposób, i co za tym idzie będziemy oznaczać użycie `read` tak, jak to robimy przy wszystkich innych operacjach, które powodują obliczenie wartości.

<sup>1</sup> Takie założenie pozwala nam pominąć mnóstwo szczegółów technicznych. Zwykle duża część implementacji systemu Lispowego jest poświęcona wczytywaniu i wypisywaniu.

Z kolei print w zasadniczy sposób różni się od operacji, których używaliśmy do tej pory — nie tworzy ona żadnej wartości, która ma być zapamiętana w rejestrze. Chociaż pociąga ona za sobą skutek, jednak skutek ten nie dotyczy części maszyny, którą projektujemy. Operacje tego typu będziemy określać jako *akcje*. Na diagramach ścieżek danych będziemy reprezentować akcje tak samo jak operacje, które powodują obliczenie wartości — w postaci trapezu zawierającego nazwę akcji. Strzałki mogą prowadzić do pudelka akcji od dowolnych danych wejściowych (rejestrów lub stałych). Akcji przyporządkowujemy również przycisk. Naciśnięcie przycisku powoduje wykonanie akcji. Do oznaczenia naciśnięcia przez sterownik przycisku akcji używamy nowego rodzaju instrukcji o nazwie *perform*. Tak więc akcja polegająca na wypisaniu zawartości rejestru a jest reprezentowana w ciągu instrukcji sterownika jako

```
(perform (op print) (reg a))
```

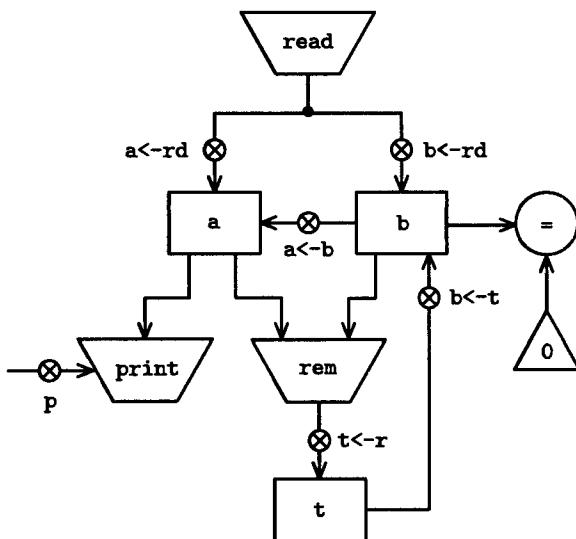
Na rysunku 5.4 są pokazane ścieżki danych i sterownik nowej maszyny obliczającej NWD. Po wypisaniu wyniku maszyna zamiast się zatrzymywała, zaczyna działanie od początku i w kółko wczytuje parę liczb, oblicza ich NWD i wypisuje wynik. Taka struktura przypomina pętlę sterującą, której używaliśmy w interpreterach z rozdziału 4.

### 5.1.2. Abstrakcja w projektach maszyn

Często będziemy określać, że maszyna zawiera operacje „pierwotne”, które faktycznie są bardzo złożone. Na przykład w podrozdziałach 5.4 i 5.5 będziemy traktować operacje na środowiskach języka Scheme jak operacje pierwotne. Taka abstrakcja jest cenna, gdyż pozwala na pomijanie szczegółów części składowych maszyny, dzięki czemu możemy się skupić na innych aspektach projektu. Fakt, że część złożoności projektu „zmietliśmy pod dywan”, nie oznacza wcale, że projekt maszyny jest nierealistyczny. Zawsze możemy zastąpić złożone operacje „pierwotne” prostszymi operacjami pierwotnymi.

Rozważmy maszynę obliczającą NWD. Maszyna ta zawiera instrukcję, która oblicza resztę z dzielenia zawartości rejestrów a i b oraz zapisuje wynik w rejestrze t. Jeśli chcemy skonstruować maszynę obliczającą NWD bez użycia operacji pierwotnej obliczającej resztę z dzielenia, musimy określić, jak obliczać reszty z dzielenia za pomocą prostszych operacji, takich jak odejmowanie. Istotnie, możemy napisać procedurę w języku Scheme, która oblicza resztę z dzielenia w następujący sposób:

```
(define (remainder n d)
  (if (< n d)
      n
      (remainder (- n d) d)))
```



```

(controller
  gcd-loop
    (assign a (op read))
    (assign b (op read))
  test-b
    (test (op =) (reg b) (const 0))
    (branch (label gcd-done))
    (assign t (op rem) (reg a) (reg b))
    (assign a (reg b))
    (assign b (reg t))
    (goto (label test-b))
  gcd-done
    (perform (op print) (reg a))
    (goto (label gcd-loop)))

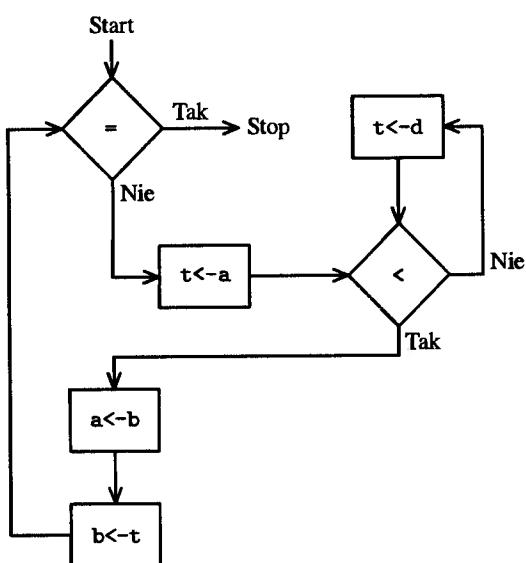
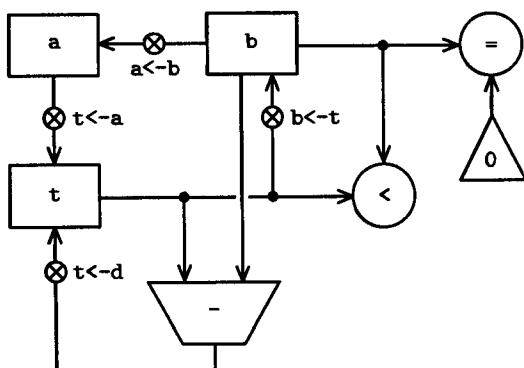
```

Rys. 5.4. Maszyna wczytująca dane, obliczającą NWD i wypisującą wyniki

Możemy więc w ścieżkach danych maszyny obliczającej NWD zastąpić operację obliczania reszty z dzielenia operacją odejmowania i porównywaniem. Na rysunku 5.5 są przedstawione ścieżki danych i sterownik tak przerobionej maszyny. Instrukcja

```
(assign t (op rem) (reg a) (reg b))
```

w definicji sterownika NWD została zastąpiona przez ciąg instrukcji zawierający pętlę, jak widać to na rys. 5.6.



Rys. 5.5. Ścieżki danych oraz sterownik przerobionej maszyny obliczającej NWD

```

(controller
  test-b
    (test (op =) (reg b) (const 0))
    (branch (label gcd-done))
    (assign t (reg a))
  rem-loop
    (test (op <) (reg t) (reg b))
    (branch (label rem-done))
    (assign t (op -) (reg t) (reg b))
    (goto (label rem-loop))
  rem-done
    (assign a (reg b))
    (assign b (reg t))
    (goto (label test-b))
  gcd-done)

```

Rys. 5.6. Ciąg instrukcji sterownika maszyny obliczającej NWD przedstawionej na rys. 5.5

### Ćwiczenie 5.3

Zaprojektuj maszynę obliczającą pierwiastki kwadratowe metodą Newtona, tak jak było to opisane w punkcie 1.1.7:

```

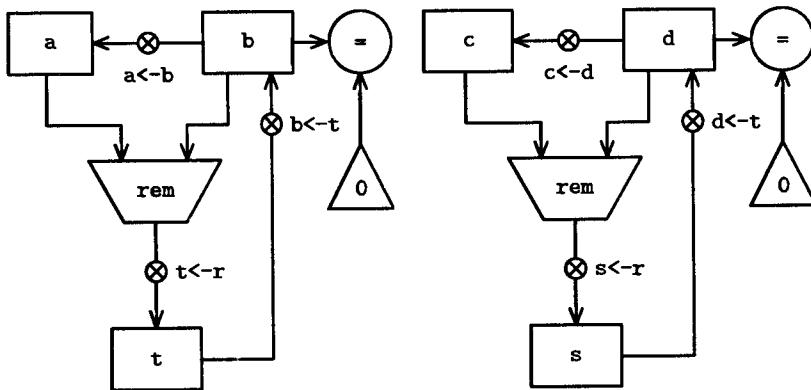
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))

```

Na początku załóż, że operacje `good-enough?` i `improve` są dostępne jako operacje pierwotne. Następnie pokaż, jak można je rozwinąć, używając operacji arytmetycznych. Każdą z wersji projektu maszyny obliczającej `sqrt` opisz, rysując diagram ścieżek danych i zapisując definicję sterownika w języku opisu maszyn rejestrowych.

#### 5.1.3. Podprogramy

Projektując maszynę, która ma wykonywać obliczenia, często będziemy tak łączyć składowe, aby współdzieliły różne części obliczeń, zamiast je powiełać. Rozważmy maszynę, która zawiera dwa obliczenia NWD — jedno, które wyznacza NWD zawartości rejestrów `a` i `b`, i drugie, które wyznacza NWD zawartości rejestrów `c` i `d`. Możemy najpierw założyć, że dostępna jest operacja pierwotna `gcd`, a następnie rozwinąć jej dwa wystąpienia za pomocą bardziej pierwotnych operacji. Na rysunku 5.7 są pokazane fragmenty dia-



```

gcd-1
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-1))
after-gcd-1
  :
gcd-2
  (test (op =) (reg d) (const 0))
  (branch (label after-gcd-2))
  (assign s (op rem) (reg c) (reg d))
  (assign c (reg d))
  (assign d (reg s))
  (goto (label gcd-2))
after-gcd-2

```

Rys. 5.7. Fragmenty diagramu ścieżek danych i ciągu instrukcji sterownika maszyny odpowiedzialne za obliczanie NWD

gramów ścieżek danych tak powstałej maszyny, odpowiedzialne za obliczanie NWD, z pominięciem ich połączenia z resztą maszyny. Na rysunku są również podane odpowiednie fragmenty ciągu instrukcji sterownika maszyny.

Maszyna ta zawiera dwa pudełka reprezentujące operacje obliczania reszty z dzielenia i dwa pudełka reprezentujące sprawdzanie równości. Jeśli powtarzane części są skomplikowane, jak w przypadku pudełka dla reszty z dzielenia, nie będzie to oszczędny sposób konstrukcji maszyny. Możemy uniknąć powtarzania składowych ścieżek danych, używając tych samych składowych w obu obliczeniach NWD, pod warunkiem że nie wpłynie to na pozostałą część obliczeń całej maszyny. Jeśli wartości w rejestrach a i b nie są potrzebne, gdy sterownik dochodzi do gcd-2 (lub jeśli wartości te mogą być zachowane w innych rejestrach), to możemy tak zmienić maszynę, aby przy

```

gcd-1
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-1))
after-gcd-1
  :
gcd-2
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-2))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-2))
after-gcd-2

```

Rys. 5.8. Fragmenty ciągu instrukcji sterownika maszyny używającej tej samej składowej ścieżek danych do dwóch różnych obliczeń NWD

obliczaniu drugiego NWD zamiast rejestrów c i d używała rejestrów a i b, tych samych co w przypadku pierwszego NWD. Jeśli tego dokonamy, to uzyskamy ciąg instrukcji sterownika pokazany na rys. 5.8.

Usunęliśmy powtarzające się składowe ścieżek danych (uzyskując ponownie ścieżki danych takie jak na rys. 5.1), ale sterownik zawiera teraz dwa ciągi instrukcji obliczania NWD, które różnią się jedynie etykietami punktów wejścia. Lepiej byłoby zastąpić te dwa ciągi instrukcji skokami do jednego ciągu — *podprogramu gcd* — po zakończeniu którego skaczemy z powrotem do odpowiedniego miejsca w głównym ciągu instrukcji. Możemy to uzyskać w następujący sposób: Przed skokiem do gcd umieszczać w specjalnym rejestrze *continue* wyróżnioną wartość (taką jak 0 lub 1). Na koniec podprogramu gcd skaczemy albo do after-gcd-1, albo do after-gcd-2, w zależności od wartości rejestrów *continue*. Na rysunku 5.9 są pokazane odnośne fragmenty tak powstałego ciągu instrukcji sterownika, który zawiera tylko jeden egzemplarz instrukcji gcd.

Jest to sensowny sposób radzenia sobie z małymi problemami, ale staje się on niewygodny, gdy w ciągu instrukcji sterownika występuje wiele obliczeń NWD. Aby rozstrzygnąć, gdzie należy kontynuować obliczenia po wykonaniu podprogramu gcd, potrzeba warunków w ścieżkach danych i instrukcji skoku sterownika dla wszystkich miejsc, w których używa się gcd. Lepszy sposób implementacji podprogramów polega na tym, że rejestr *continue* zawiera etykietę punktu wejścia do ciągu instrukcji sterownika, od którego obliczenia powinny być kontynuowane po zakończeniu podprogramu. Zaimplementowanie tej techniki wymaga nowego rodzaju połączenia między ścieżkami danych

```

gcd
  (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd))
gcd-done
  (test (op =) (reg continue) (const 0))
  (branch (label after-gcd-1))
  (goto (label after-gcd-2))
  :
;; Przed skokiem do gcd z pierwszego miejsca, w którym jest
;; ona potrzebna, umieszczamy w rejestrze continue wartość 0.
  (assign continue (const 0))
  (goto (label gcd))
after-gcd-1
  :
;; Przed drugim użyciem gcd umieszczamy w rejestrze continue wartość 1.
  (assign continue (const 1))
  (goto (label gcd))
after-gcd-2

```

Rys. 5.9. Użycie rejestrów `continue` w celu uniknięcia powtarzających się ciągów instrukcji z rys. 5.8

i sterownikiem maszyny rejestrowej — musi istnieć możliwość przypisania rejestrów etykiety pochodzącej z ciągu instrukcji sterownika w taki sposób, aby jej wartość mogła być wydobyta z rejestrów i użyta do kontynuowania obliczeń od określonego punktu wejścia.

Aby odzwierciedlić tę możliwość, rozszerzymy instrukcję `assign` języka opisu maszyn rejestrowych tak, żeby umożliwiała przypisywanie rejestrów etykiet pochodzących z ciągu instrukcji sterownika (jako stałych specjalnego rodzaju). Rozszerzymy również instrukcję `goto`, tak aby umożliwiała kontynuowanie wykonywania od punktu wejścia określonego przez zawartość rejestrów, a nie tylko od punktów wejścia opisywanych przez stałe etykiety. Za pomocą tych nowych instrukcji możemy zakończyć podprogram `gcd` skokiem do miejsca zapamiętanego w rejestrze `continue`. Prowadzi to do ciągu instrukcji sterownika pokazanego na rys. 5.10.

Maszyna z więcej niż jednym podprogramem mogłaby używać wielu rejestrów kontynuacji (np. `gcd-continue`, `factorial-continue` itd.) lub też wszystkie podprogramy mogłyby współdzielić jeden rejestr `continue`. Współdzielenie jest bardziej oszczędne, ale musimy uważać, jeżeli mamy podprogram (`sub1`), który wywołuje inny podprogram (`sub2`). Jeżeli `sub1` nie zachowa wartości `continue` w jakimś innym rejestrze przed ustawieniem `continue` na

```

gcd
  (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd))
gcd-done
  (goto (reg continue))
  :
  ;: Przed wywołaniem gcd rejestrowi continue
  ;: przypisujemy etykietę, do której gcd powinno powrócić.
  (assign continue (label after-gcd-1))
  (goto (label gcd))
after-gcd-1
  :
  ;: Drugie wywołanie gcd, z inną kontynuacją obliczeń.
  (assign continue (label after-gcd-2))
  (goto (label gcd))
after-gcd-2

```

Rys. 5.10. Przypisywanie etykiet rejestrowi `continue` upraszcza i uogólnia technikę pokazaną na rys. 5.9

potrzeby wywołania `sub2`, to nie będzie wiadomo, gdzie należy skoczyć po zakończeniu `sub1`. Mechanizm, który przedstawiamy w następnym punkcie, służący do obsługi rekursji stanowi również lepsze rozwiązanie problemu zagnieżdżonych wywołań podprogramów.

#### 5.1.4. Implementacja rekursji za pomocą stosu

Z pomocą przedstawionych dotychczas koncepcji możemy zaimplementować dowolny proces iteracyjny, określając maszynę rejestrową, która dla każdej zmiennej stanu zawiera odpowiadający jej rejestr. Maszyna ta wielokrotnie wykonuje pętlę sterownika, zmieniając zawartość rejestrów, aż do spełnienia pewnego warunku końcowego. W każdym punkcie ciągu instrukcji sterownika stan maszyny (reprezentujący stan procesu iteracyjnego) jest całkowicie określony przez zawartość rejestrów (wartości zmiennych stanu).

Jednakże implementacja procesów rekurencyjnych wymaga dodatkowego mechanizmu. Rozważmy następującą rekurencyjną metodę obliczania silni, którą po raz pierwszy zajmowaliśmy się w punkcie 1.2.1:

```

(define (factorial n)
  (if (= n 1)
    1
    (* (factorial (- n 1)) n)))

```

Jak widać z procedury, obliczenie  $n!$  wymaga obliczenia  $(n-1)!$ . Podobnie nasza maszyna obliczająca NWD, wzorowana na procedurze

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

musi obliczyć inne NWD. Jest jednak istotna różnica między procedurą `gcd`, która sprowadza obliczenie początkowego NWD do obliczenia innego NWD, a procedurą `factorial`, która wymaga obliczenia innej silni jako podproblemu. W przypadku obliczania NWD, wynik nowego obliczenia NWD jest równocześnie wynikiem początkowego problemu. Chcąc obliczyć kolejne NWD, umieszczaamy po prostu nowe argumenty w rejestrach wejściowych maszyny obliczającej NWD i ponownie używamy tych samych ścieżek danych maszyny, wykonując ten sam ciąg instrukcji sterownika. Gdy maszyna zakończy rozwiązywanie ostatniego problemu obliczania NWD, zakończy tym samym całe obliczenie.

W przypadku obliczania silni (bądź wykonywania dowolnego procesu rekurencyjnego) wynik nowego podproblemu obliczenia silni nie jest wynikiem początkowego problemu. Wartość będąca wynikiem  $(n-1)!$  musi być jeszcze pomnożona przez  $n$ , aby otrzymać ostateczny wynik. Jeśli spróbujemy naśladować konstrukcję obliczania NWD i rozwiążemy podproblem obliczania silni, zmniejszając wartość w rejestrze `n` i ponownie uruchamiając maszynę obliczającą silnię, to nie będziemy już dysponowali starą wartością `n`, przez którą należy pomnożyć wynik. Dlatego też potrzebujemy drugiej maszyny do rozwiązywania podproblemu. Jednak to drugie obliczenie silni zawiera podproblem obliczania silni, który wymaga trzeciej maszyny itd. Ponieważ każda maszyna obliczająca silnię zawiera w sobie kolejną maszynę obliczającą silnię, więc cała maszyna zawiera nieskończonie wiele zagnieżdżeń podobnych maszyn i w związku z tym nie może być zbudowana z ustalonej, skończonej liczby części.

Niemniej jednak możemy zaimplementować proces obliczający silnię jako maszynę rejestrową, jeśli zdołamy użyć tych samych części składowych dla każdego zagnieżdzonego egzemplarza maszyny. Mówiąc ściślej, maszyna obliczająca  $n!$  powinna używać tych samych części do podproblemu obliczenia  $(n-1)!$ , podproblemu  $(n-2)!$  itd. Jest to możliwe, gdyż mimo że proces obliczania silni wymusza na potrzeby wykonania obliczenia istnienie nieograniczonej liczby kopii tej samej maszyny, jednak w każdej chwili jest używana tylko jedna z tych kopii. Gdy maszyna napotyka na podproblem rekurencyjny, wstrzymuje obliczenie głównego problemu, ponownie korzysta z tych samych fizycznych części do rozwiązywania podproblemu, po czym kontynuuje wstrzymane obliczenie.

Zawartość rejestrów przy rozwiązywaniu podproblemu będzie inna niż przy rozwiązywaniu głównego problemu. (W tym przypadku zmniejszana jest wartość w rejestrze  $n$ ). Aby móc kontynuować wstrzymane obliczenie, maszyna musi zachować zawartość wszystkich rejestrów, które będą potrzebne po rozwiązaniu podproblemu, tak aby móc przywrócić ich zawartość i kontynuować wstrzymane obliczenie. W przypadku obliczania silni zachowamy starą wartość  $n$ , aby móc ją przywrócić, gdy skończymy obliczanie silni dla zmniejszonej wartości w rejestrze  $n$ <sup>2</sup>.

Ponieważ nie istnieje żadne ograniczenie a priori na głębokość zagnieźdzenia wywołań rekurencyjnych, możemy być zmuszeni zapamiętać wartości dowolnej liczby rejestrów. Wartości te muszą być odtwarzane w odwrotnej kolejności, niż były zapamiętywane, gdyż przy zagnieźdzanej rekurencji podproblem, który zaczynamy jako ostatni, jest pierwszym, który zostanie obliczony. Wymaga to użycia do zapamiętywania wartości rejestrów *stosu* bądź struktury danych typu „ostatni na wejściu — pierwszy na wyjściu”. Możemy rozszerzyć język opisu maszyn rejestrowych o stos, dodając dwa rodzaje instrukcji: wartości można zapamiętywać na stosie za pomocą instrukcji *save* (zachowaj) i odtwarzać je ze stosu za pomocą instrukcji *restore* (odtwarz). Po zapamiętaniu wartości na stosie za pomocą operacji *save* odtwarzamy je ze stosu za pomocą operacji *restore* w odwrotnej kolejności<sup>3</sup>.

Używając stosu, możemy wykorzystać tę samą kopię ścieżek danych maszyny obliczającej silnię do obliczania każdego z podproblemów silni. Podobna kwestia dotyczy wielokrotnego użycia tego samego ciągu instrukcji sterownika operujących na ścieżkach danych. Aby ponownie rozpoczęć obliczanie silni, sterownik nie może po prostu skoczyć do początku ciągu instrukcji, jak to miało miejsce w przypadku procesu iteracyjnego, gdyż po rozwiązaniu podproblemu  $(n - 1)!$  maszyna musi jeszcze pomnożyć wynik przez  $n$ . Sterownik musi wstrzymać obliczanie  $n!$ , rozwiązać podproblem  $(n - 1)!$ , a następnie wznowić obliczanie  $n!$ . Takie spojrzenie na obliczanie silni wskazuje na użycie mechanizmu podprogramów, opisanego w punkcie 5.1.3, w którym sterownik korzysta z rejestru *continue* do zapamiętania miejsca w głównym problemie, z którego następuje skok do ciągu instrukcji rozwiązywających podproblem, a następnie powrotu do głównego problemu. Możemy więc tak zaimplementować podprogram obliczania silni, aby wracał do punktu wejścia zapamiętanego

<sup>2</sup> Można by utrzymywać, że nie jest konieczne zapamiętywanie starej wartości  $n$  — po zmniejszeniu wartości w tym rejestrze i rozwiązaniu podproblemu można by po prostu zwiększyć tę wartość i odzyskać starą wartość. Choć takie rozwiązanie działa w przypadku silni, jednak nie działa ono w ogólności, gdyż stara zawartość rejestru nie zawsze może być obliczona na podstawie nowej.

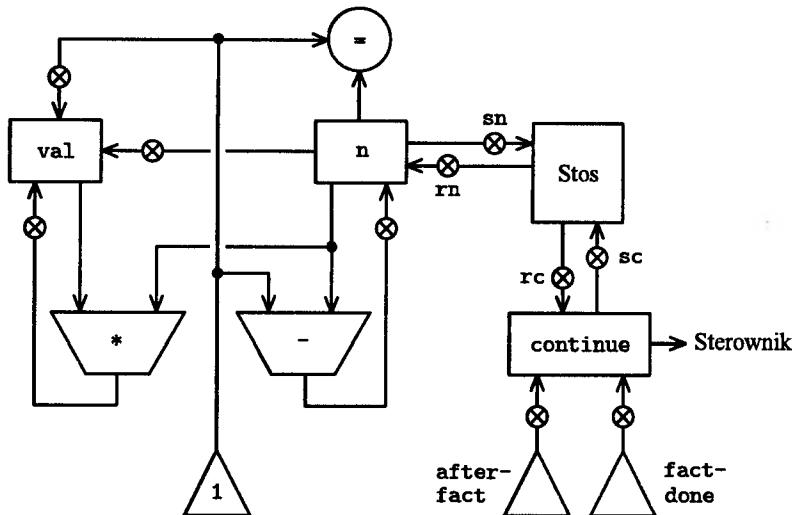
<sup>3</sup> Zwykle operacje te są nazywane *push* i *pop* (przyp. tłum.).

<sup>3</sup> W podrozdziale 5.3 zobaczymy, jak można zaimplementować stos za pomocą prostszych operacji.

w rejestrze `continue`. Przed każdym wywołaniem podprogramu zachowujemy, a po powrocie odtwarzamy zawartość rejestrów `continue`, tak samo jak to robimy z rejestrów `n`, ponieważ na każdym „poziomie” obliczeń silni będzie używany ten sam rejestr `continue`. Oznacza to, że podprogram obliczania silni, gdy wywołuje sam siebie w celu rozwiązyania podproblemu, musi przypisać rejestrowi `continue` nową wartość, ale będzie potrzebował jego starej wartości, aby wrócić do miejsca, z którego został wywołany.

Na rysunku 5.11 widać ścieżki danych i sterownik maszyny implementującej rekurencyjną procedurę `factorial`. Maszyna ta ma stos oraz trzy rejestrów o nazwach `n`, `val` i `continue`. Aby uprościć diagram ścieżek danych, nie nazwaliśmy przycisków przypisujących wartości rejestrów, a jedynie przyciski operacji na stosie (`sc` i `sn` zachowują rejestr, a `rc` i `rn` odtwarzają je). Aby maszyna zaczęła działać, zapisujemy w rejestrze `n` liczbę, której silni chcemy obliczyć, i uruchamiamy maszynę. Gdy maszyna dochodzi do etykiety `fact-done`, obliczenie jest zakończone, a wynik znajduje się w rejestrze `val`. W ciągu instrukcji sterownika zawartości rejestrów `n` i `continue` są zachowywane przed każdym wywołaniem rekurencyjnym, po czym są odtwarzane po każdym powrocie z wywołania. Powrót z wywołania jest realizowany jako skok do miejsca zapamiętanego w rejestrze `continue`. Zawartość rejestrów `continue` jest inicjowana w momencie uruchomienia maszyny w ten sposób, że ostatni powrót spowoduje skok do etykiety `fact-done`. Rejestr `val`, który zawiera wynik obliczenia silni, nie jest zachowywany przed wywołaniem rekurencyjnym, gdyż jego stara zawartość nie jest używana po powrocie z podprogramu. Potrzebna jest tylko jego nowa wartość, która jest równa wynikowi podobliczenia.

Choć w zasadzie obliczenie silni wymaga maszyny nieskończonej, maszyna przedstawiona na rys. 5.11 jest w rzeczywistości skończona, z wyjątkiem stosu, który jest potencjalnie nieograniczony. Jednak każda konkretna fizyczna implementacja stosu będzie skończona, co będzie ograniczać głębokość wywołań rekurencyjnych, jakie mogą być wykonywane przez maszynę. Przedstawiona tutaj implementacja obliczania silni ilustruje ogólną technikę realizacji algorytmów rekurencyjnych w postaci zwykłych maszyn rejestrów z dodanymi stosami. Gdy napotykamy podproblem rekurencyjny, zapamiętujemy na stosie rejestr, których bieżące wartości będą potrzebne po rozwiązyaniu podproblemu, rozwiązujemy podproblem rekurencyjny, po czym odtwarzamy zapamiętane rejestr i kontynuujemy rozwiązywanie głównego problemu. Rejestr `continue` musi być zawsze zapamiętywany. To, czy inne rejestrury muszą być zapamiętywane, zależy od konkretnej maszyny, gdyż nie we wszystkich obliczeniach rekurencyjnych potrzebne są początkowe wartości rejestrów, które są modyfikowane w trakcie rozwiązywania podproblemu (zob. ćwiczenie 5.4).



```

(controller
  (assign continue (label fact-done)) ; końcowy adres powrotu
fact-loop
  (test (op =) (reg n) (const 1))
  (branch (label base-case))
;; Zachowaj przed wywołaniem rekurencyjnym rejestr n i continue.
;; Ustaw rejestr continue tak, aby po powrocie z podprogramu
;; obliczenie było kontynuowane od etykiety after-fact.
  (save continue)
  (save n)
  (assign n (op -) (reg n) (const 1))
  (assign continue (label after-fact))
  (goto (label fact-loop))
after-fact
  (restore n)
  (restore continue)
  (assign val (op *) (reg n) (reg val)) ; val jest teraz równe  $n(n - 1)!$ 
  (goto (reg continue)) ; powrót do miejsca wywołania
base-case
  (assign val (const 1)) ; przypadek podstawowy:  $1! = 1$ 
  (goto (reg continue)) ; powrót do miejsca wywołania
fact-done)

```

Rys. 5.11. Rekurencyjna maszyna obliczająca silnię

### Dwukrotna rekursja

Zbadajmy bardziej złożony proces rekurencyjny, a mianowicie rozgałęziający się rekurencyjnie proces obliczania liczb Fibonacciego, który przedstawiliśmy w punkcie 1.2.2:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Tak jak w przypadku obliczania silni, możemy zaimplementować rekurencyjne obliczanie liczb Fibonacciego w postaci maszyny rejestrowej z rejestrami  $n$ ,  $val$  i  $continue$ . Maszyna ta jest bardziej skomplikowana niż w przypadku silni, gdyż wywołania rekurencyjne występują w dwóch miejscach w ciągu instrukcji sterownika — gdy obliczamy  $Fib(n-1)$  i gdy obliczamy  $Fib(n-2)$ . Przed każdym z tych wywołań zachowujemy rejesty, których wartości będą potrzebne później, ustawiamy wartość rejestrów  $n$  na numer liczby Fibonacciego, którą chcemy rekurencyjnie obliczyć ( $n-1$  lub  $n-2$ ), i przypisujemy rejestrowi  $continue$  punkt wejścia w głównym ciągu instrukcji, do którego należy powrócić (odpowiednio `afterfib-n-1` lub `afterfib-n-2`). Następnie skaczemy do `fib-loop`. Gdy następuje powrót z wywołania rekurencyjnego, wynik znajduje się w rejestrze  $val$ . Ciąg instrukcji sterownika takiej maszyny jest przedstawiony na rys. 5.12.

### Ćwiczenie 5.4

Wyspecyfikuj maszyny rejestrowe implementujące obie poniższe procedury. Dla każdej maszyny podaj ciąg instrukcji sterownika i narysuj diagram ścieżek danych.

(a) Potęgowanie rekurencyjne:

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

(b) Potęgowanie iteracyjne:

```
(define (expt b n)
  (define (expt-iter counter product)
    (if (= counter 0)
        product
        (expt-iter (- counter 1) (* b product))))
  (expt-iter n 1))
```

### Ćwiczenie 5.5

Zasymuluj ręcznie maszyny obliczające silnię i liczby Fibonacciego dla jakiś nietrywialnych danych (wymagających wykonania przynajmniej jednego wywołania rekurencyjnego). Dla każdego istotnego momentu wykonania pokaż zawartość stosu.

```

(controller
  (assign continue (label fib-done))
  fib-loop
    (test (op <) (reg n) (const 2))
    (branch (label immediate-answer))
    ;; oblicz Fib(n - 1)
    (save continue)
    (assign continue (label afterfib-n-1))
    (save n) ; zachowaj starą wartość n
    (assign n (op -) (reg n) (const 1)) ; zmniejsz n o 1
    (goto (label fib-loop)) ; wywołanie rekurencyjne
  afterfib-n-1 ; po powrocie val jest równe Fib(n - 1)
    (restore n)
    (restore continue)
    ;; oblicz Fib(n - 2)
    (assign n (op -) (reg n) (const 2))
    (save continue)
    (assign continue (label afterfib-n-2))
    (save val) ; zachowaj Fib(n - 1)
    (goto (label fib-loop))
  afterfib-n-2 ; po powrocie val jest równe Fib(n - 2)
    (assign n (reg val)) ; n jest teraz równe Fib(n - 2)
    (restore val) ; val jest teraz równe Fib(n - 1)
    (restore continue)
    (assign val ; Fib(n - 1) + Fib(n - 2)
      (op +) (reg val) (reg n))
    (goto (reg continue)) ; powrót, wynik jest w val
  immediate-answer ; przypadek podstawowy: Fib(n) = n
    (assign val (reg n))
    (goto (reg continue))
  fib-done)

```

Rys. 5.12. Sterownik maszyny obliczającej liczby Fibonacciego

### Ćwiczenie 5.6

Ben Bajerbit zauważał, że ciąg instrukcji sterownika maszyny obliczającej liczby Fibonacciego zawiera po jednej dodatkowej instrukcji `save` i `restore`, które można usunąć, przyspieszając działanie maszyny. Które to instrukcje?

#### 5.1.5. Podsumowanie instrukcji

Instrukcje sterownika w naszym języku opisu maszyn rejestrowych są jednej z poniższych postaci, gdzie każde  $\langle arg_i \rangle$  to albo `(reg nazwa rejestr)`, albo `(const stała)`.

Następujące instrukcje zostały wprowadzone w punkcie 5.1.1:

```

(assign <nazwa rejestr> (reg <nazwa rejestr>))
(assign <nazwa rejestr> (const <stała>))

```

```
(assign <nazwa rejestr> (op <nazwa operacji>) <arg1> ... <argn>)
(perform (op <nazwa operacji>) <arg1> ... <argn>)
(test (op <nazwa operacji>) <arg1> ... <argn>)
(branch (label <nazwa etykiety>))
(goto (label <nazwa etykiety>))
```

Możliwość przechowywania etykiet w rejestrach była wprowadzona w punkcie 5.1.3:

```
(assign <nazwa rejestr> (label <nazwa etykiety>))
(goto (reg <nazwa rejestr>))
```

Operacje na stosie były opisane w punkcie 5.1.4:

```
(save <nazwa rejestr>)
	restore <nazwa rejestr>)
```

Jedyny rodzaj *stałych*, jakie widzieliśmy do tej pory, to liczby, ale dalej będziemy używać również napisów, symboli i list. Na przykład (const "abc") to napis "abc", (const abc) to symbol abc, (const (a b c)) to lista (a b c), a (const ()) to lista pusta.

## 5.2. Symulator maszyny rejestrowej

Chcąc dobrze zrozumieć projektowanie maszyn rejestrowych, musimy być w stanie testować projektowane przez nas maszyny, aby sprawdzić, czy działają tak, jak tego oczekujemy. Jeden ze sposobów testowania projektów polega na ręcznym symulowaniu operacji sterownika, jak to robiliśmy w ćwiczeniu 5.5. Jednak, pominąwszy najprostsze maszyny, jest to bardzo nużące. W niniejszym podrozdziale konstruujemy symulator maszyn opisywanych w języku maszyn rejestrowych. Symulator ten jest programem w języku Scheme, którego interfejs składa się z czterech procedur. Pierwsza z nich na podstawie opisu maszyny rejestrowej buduje jej model (strukturę danych, której elementy odpowiadają składowym symulowanej maszyny), a pozostałe trzy służą do symulowania maszyny, operując na jej modelu:

```
(make-machine <nazwy rejestrów> <operacje> <sterownik>)
konstruuje model maszyny zawierającej podane rejestr, operacje i sterownik;
jej wynikiem jest utworzony model.

(set-register-contents! <model maszyny> <nazwa rejestr> <wartość>)
zapamiętuje podaną wartość w określonym rejestrze modelowanej maszyny.
```

(*get-register-contents* *<model maszyny>* *<nazwa rejestrów>*)  
 daje w wyniku zawartość określonego rejestrów modelowanej maszyny.

(*start* *<model maszyny>*)

symuluje działanie danej maszyny, zaczynając wykonywanie instrukcji sterownika od początku ciągu i kończąc, gdy sterowanie dochodzi do końca ciągu instrukcji.

Aby zilustrować użycie tych procedur, możemy w następujący sposób zdefiniować model gcd-machine maszyny obliczającej NWD, przedstawionej w punkcie 5.1.1:

```
(define gcd-machine
  (make-machine
    '(a b t)
    (list (list 'rem remainder) (list '= =))
    '(test-b
      (test (op =) (reg b) (const 0))
      (branch (label gcd-done))
      (assign t (op rem) (reg a) (reg b))
      (assign a (reg b))
      (assign b (reg t))
      (goto (label test-b))
    gcd-done)))
```

Pierwszy argument procedury *make-machine* to lista nazw rejestrów. Kolejny argument to tablica (lista dwuelementowych list), która łączy każdą nazwę operacji z implementującą ją procedurą w języku Scheme (tzn. taką procedurą, która dla takich samych argumentów daje takie same wyniki). Ostatni argument opisuje sterownik za pomocą listy etykiet i instrukcji maszynowych, tak jak w podrozdziale 5.1.

Chcąc obliczyć za pomocą tej maszyny NWD, ustawiamy wartości rejestrów wejściowych, uruchamiamy maszynę i gdy symulacja się zatrzyma, badamy wyniki:

```
(set-register-contents! gcd-machine 'a 206)
done

(set-register-contents! gcd-machine 'b 40)
done

(start gcd-machine)
done

(get-register-contents gcd-machine 'a)
2
```

Obliczenie to będzie wykonywane dużo wolniej niż procedura gcd napisana w języku Scheme, ponieważ instrukcje niskopoziomowe, takie jak `assign`, symulujemy za pomocą dużo bardziej złożonych operacji.

### Ćwiczenie 5.7

Przetestuj za pomocą symulatora maszyny, które zaprojektowałeś w ćwiczeniu 5.4.

#### 5.2.1. Model maszyny

Model maszyny generowany przez `make-machine` jest reprezentowany jako procedura ze stanem lokalnym, przy użyciu techniki przekazywania komunikatów opracowanej w rozdziale 3. Aby zbudować taki model, `make-machine` zaczyna od wywołania procedury `make-new-machine`, żeby utworzyć elementy modelu, które występują we wszystkich maszynach rejestrowych. Taki podstawowy model maszyny tworzony przez `make-new-machine` to w gruncie rzeczy pojemnik na rejesty, stos i mechanizm, który przetwarza, jedną po drugiej, instrukcje sterownika.

Następnie `make-machine` rozszerza ten podstawowy model (wysyłając do niego komunikaty) o rejesty, operacje i sterownik konkretnej, definiowanej maszyny. Najpierw dla każdej nazwy rejestrów jest przydzielany w nowej maszynie rejestr i są w niej instalowane podane operacje. Potem za pomocą *assemblera* (opisanego w punkcie 5.2.2) `make-machine` przekształca listę instrukcji sterownika w instrukcje dla nowej maszyny i instaluje je. Wynikiem `make-machine` jest zmodyfikowany model maszyny.

```
(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each (lambda (register-name)
                ((machine 'allocate-register) register-name))
              register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

### Rejestry

Rejestry będziemy reprezentować w postaci procedur ze stanem lokalnym, jak to robiliśmy w rozdziale 3. Procedura `make-register` tworzy rejestr zawierający wartość, która jest dostępna i którą można zmieniać:

```
(define (make-register name)
  (let ((contents '*unassigned*))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value)
               (set! contents value)
               contents))))))
```

```

  (lambda (value) (set! contents value)))
  (else
    (error "Nieznane polecenie -- REGISTER" message)))
  dispatch))

```

Następujące procedury udostępniają zawartości rejestrów:

```

(define (get-contents register)
  (register 'get))

(define (set-contents! register value)
  ((register 'set) value))

```

### Stos

Stos możemy również reprezentować w postaci procedury ze stanem lokalnym. Procedura `make-stack` tworzy stos, którego stan lokalny zawiera listę elementów na stosie. Stos realizuje zlecenia wkładania (`push`) elementów na stos oraz zdejmowania (`pop`) ich z wierzchołka stosu i przekazywania jako wyników, a także inicjowania (`initialize`) pustego stosu.

```

(define (make-stack)
  (let ((s '()))
    (define (push x)
      (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Pusty stos -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top)))
    (define (initialize)
      (set! s '())
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Nieznane polecenie -- STACK"
                         message))))
    dispatch)))

```

Następujące procedury realizują dostęp do stosów:

```

(define (pop stack)
  (stack 'pop))

(define (push stack value)
  ((stack 'push) value))

```

### Maszyna podstawowa

Procedura `make-new-machine`, przedstawiona na rys. 5.13, tworzy obiekt, którego stan lokalny składa się ze stosu, początkowo pustego ciągu instrukcji, listy operacji, która początkowo zawiera operację inicjującą stos, oraz *tablicy rejestrów* (ang. *register table*), która początkowo zawiera dwa rejestrów o nazwach `flag` i `pc` (od ang. *program counter* — licznik rozkazów). Procedura wewnętrzna `allocate-register` dodaje nowe pozycje w tablicy rejestrów, a procedura wewnętrzna `lookup-register` sprawdza zawartość rejestrów w tablicy.

Rejestr `flag` służy do sterowania skokami warunkowymi w symulowanej maszynie. Instrukcja `test` zapisuje w tym rejestrze wyniki sprawdzanych warunków (prawdę lub fałsz). Instrukcja `branch` decyduje, czy wykonywać skok na podstawie zawartości rejestrów `flag`.

Rejestr `pc` służy do określania w trakcie działania programu kolejnej instrukcji do wykonania. Kolejność wykonywania instrukcji jest zaimplementowana przez procedurę wewnętrzna `execute`. W modelu symulacyjnym każda instrukcja maszyny stanowi strukturę danych, w skład której wchodzi procedura bezargumentowa nazywana *procedurą wykonawczą instrukcji* (ang. *instruction execution procedure*), której wywołanie symuluje wykonanie instrukcji. W trakcie symulacji rejestr `pc` wskazuje na miejsce w ciągu instrukcji, w którym znajduje się kolejna instrukcja do wykonania. Procedura `execute` pobiera tę instrukcję, wykonuje ją, wywołując jej procedurę wykonawczą, i powtarza ten cykl dopóki nie zabraknie dalszych instrukcji do wykonania (tzn. aż `pc` będzie wskazywać na koniec ciągu instrukcji).

W ramach swojego działania każda procedura wykonawcza instrukcji modyfikuje rejestr `pc` tak, aby wskazywał na kolejną instrukcję do wykonania. Instrukcje `branch` i `goto` zmieniają `pc` tak, aby wskazywał na miejsca docelowe skoków. Wszystkie pozostałe instrukcje po prostu przesuwają `pc`, tak aby wskazywał na kolejną instrukcję w ciągu. Zauważmy, że każde wywołanie `execute` powoduje kolejne wywołanie `execute`, jednak nie powoduje to wejścia w pętlę nieskończoną (zapołtenia), ponieważ uruchomienie procedury wykonawczej instrukcji zmienia zawartość rejestrów `pc`.

Wynikiem `make-new-machine` jest procedura `dispatch`, która za pomocą mechanizmu przekazywania komunikatów implementuje dostęp do stanu wewnętrznego. Zwróćmy uwagę, że uruchomienie maszyny jest realizowane przez ustawienie `pc` na początek ciągu instrukcji i wywołanie procedury `execute`.

Dla wygody udostępniamy alternatywny, proceduralny interfejs operacji `start` uruchamiającej maszynę, a także procedury służące do ustawiania i bieżania zawartości rejestrów, zgodnie z tym, co określiliśmy na początku podrozdziału 5.2:

```

(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
           (list (list 'initialize-stack
                      (lambda () (stack 'initialize)))))
          (register-table
            (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Wielokrotnie zdefiniowany rejestr:" name)
            (set! register-table
                  (cons (list name (make-register name))
                        register-table)))
        'register-allocated)
      (define (lookup-register name)
        (let ((val (assoc name register-table)))
          (if val
              (cadr val)
              (error "Nieznaný rejestr:" name))))
      (define (execute)
        (let ((insts (get-contents pc)))
          (if (null? insts)
              'done
              (begin
                ((instruction-execution-proc (car insts))
                 (execute)))))
      (define (dispatch message)
        (cond ((eq? message 'start)
               (set-contents! pc the-instruction-sequence)
               (execute))
              ((eq? message 'install-instruction-sequence)
               (lambda (seq) (set! the-instruction-sequence seq)))
              ((eq? message 'allocate-register) allocate-register)
              ((eq? message 'get-register) lookup-register)
              ((eq? message 'install-operations)
               (lambda (ops) (set! the-ops (append the-ops ops))))
              ((eq? message 'stack) stack)
              ((eq? message 'operations) the-ops)
              (else (error "Nieznané polecenie -- MACHINE" message))))
        dispatch)))

```

Rys. 5.13. Procedura `make-new-machine` implementująca podstawowy model maszyny

```
(define (start machine)
  (machine 'start))

(define (get-register-contents machine register-name)
  (get-contents (get-register machine register-name)))

(define (set-register-contents! machine register-name value)
  (set-contents! (get-register machine register-name) value)
  'done)
```

Procedury te (a także wiele procedur w punktach 5.2.2 i 5.2.3) używają następującej procedury do sprawdzenia zawartości rejestrów o podanej nazwie w danej maszynie:

```
(define (get-register machine reg-name)
  ((machine 'get-register) reg-name))
```

### 5.2.2. Asembler

Asembler przekształca ciąg wyrażeń sterujących maszyny w odpowiadającą mu listę instrukcji maszynowych, z których każda ma własną procedurę wykonawczą. Ogólnie mówiąc, assembler w dużym stopniu przypomina evaluatorów, które badaliśmy w rozdziale 4 — dany jest język wejściowy (w tym przypadku język opisu maszyn rejestrowych) i dla każdego rodzaju wyrażeń w języku należy wykonać odpowiednie działania.

Technika polegająca na tworzeniu procedur wykonawczych dla poszczególnych instrukcji to ta sama technika, którą stosowaliśmy w punkcie 4.1.7 do przyspieszenia działania evaluatora przez oddzielenie analizy od wykonywania. Jak widzieliśmy w rozdziale 4, wiele użytecznych aspektów wyrażeń w języku Scheme można zanalizować, nie znając faktycznych wartości zmiennych. Tutaj, analogicznie, wiele użytecznych aspektów wyrażeń języka opisu maszyn rejestrowych można zanalizować, nie znając faktycznych zawartości rejestrów maszynowych. Możemy na przykład zastąpić odwołania do rejestrów wskaźnikami do obiektów reprezentujących rejestrory oraz możemy zastąpić odwołania do etykiet wskaźnikami do miejsc w ciągu instrukcji, które te etykiety wyznaczają.

Zanim assembler będzie mógł wygenerować procedury wykonawcze instrukcji, musi wiedzieć, do czego odnoszą się wszystkie etykiety; zaczyna więc działanie od przejrzenia tekstu sterownika w celu oddzielenia etykiet od instrukcji. W trakcie przeglądania tekstu tworzy on listę instrukcji oraz tablicę przyporządkowującą każdej etykiecie wskaźnik do tej listy. Następnie assembler rozszerza listę, dodając do każdej instrukcji jej procedurę wykonawczą.

Procedura `assemble` stanowi główne wejście do assemblera. Jej argumentami są tekst sterownika i model maszyny, a jej wynikiem jest ciąg instrukcji, który należy zachować w modelu. `Assemble` wywołuje procedurę `extract-`

`-labels`, która na podstawie dostarczonego tekstu sterownika tworzy początkową listę instrukcji oraz tablicę etykiet. Drugim argumentem `extract-labels` jest procedura, którą należy wywołać w celu przetworzenia tych wyników — procedura ta używa `update-insts!` do wygenerowania procedur wykonawczych dla instrukcji i wstawienia ich do listy instrukcji, a jej wynikiem jest zmodyfikowana lista.

```
(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      insts)))
```

Argumentami `extract-labels` są lista `text` (ciąg wyrażeń instrukcji sterownika) oraz procedura `receive`. Z kolei procedura `receive` jest wywoływana z dwoma argumentami: (1) listą `insts` struktur danych instrukcji, z których każda zawiera instrukcję z listy `text`, oraz (2) tablicą o nazwie `labels`, która wiąże każdą etykietę z listy `text` z wyznaczaną przez nią pozycją na liście `insts`.

```
(define (extract-labels text receive)
  (if (null? text)
    (receive '() '())
    (extract-labels (cdr text)
      (lambda (insts labels)
        (let ((next-inst (car text)))
          (if (symbol? next-inst)
            (receive insts
              (cons (make-label-entry next-inst
                insts)
                labels))
            (receive (cons (make-instruction next-inst)
              insts)
              labels)))))))
```

Procedura `extract-labels` przegląda kolejne elementy listy `text` i kumuluje instrukcje `insts` oraz etykiety `labels`. Jeśli dany element jest symbolem (a więc etykietą), to odpowiednia pozycja jest wstawiana do tablicy `labels`. W przeciwnym razie element jest kumulowany na liście `insts`<sup>4</sup>.

<sup>4</sup> Użycie procedury `receive` to sposób na to, aby wynikiem `extract-labels` była faktycznie para wartości — `labels` i `insts` — bez konieczności jawnego użycia zawierającej je złożonej struktury danych. Oto alternatywna implementacja, która jawnie przekazuje jako wynik parę wartości:

```
(define (extract-labels text)
  (if (null? text)
```

Procedura `update-insts!` tak modyfikuje listę instrukcji, która początkowo zawiera tylko tekst instrukcji, aby zawierała odpowiadające im procedury wykonawcze:

```
(define (update-insts! insts labels machine)
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag))
        (stack (machine 'stack))
        (ops (machine 'operations)))
    (for-each
      (lambda (inst)
        (set-instruction-execution-proc!
          inst
          (make-execution-procedure
            (instruction-text inst) labels machine
            pc flag stack ops)))
      insts)))
```

Struktura danych instrukcji maszynowej to po prostu para złożona z tekstu instrukcji oraz odpowiedniej procedury wykonawczej. Procedura wykonawcza nie jest jeszcze dostępna, gdy `extract-labels` tworzy instrukcje, i jest wstawiana później przez `update-insts!`.

```
(define (make-instruction text)
  (cons text '()))

(define (instruction-text inst)
  (car inst))



---


(cons '() '())
(let ((result (extract-labels (cdr text))))
  (let ((insts (car result)) (labels (cdr result)))
    (let ((next-inst (car text)))
      (if (symbol? next-inst)
          (cons insts
                (cons (make-label-entry next-inst insts) labels))
          (cons (cons (make-instruction next-inst) insts)
                labels))))))
```

Byłaby ona wywoływana przez `assemble` w następujący sposób:

```
(define (assemble controller-text machine)
  (let ((result (extract-labels controller-text)))
    (let ((insts (car result)) (labels (cdr result)))
      (update-insts! insts labels machine)
      insts)))
```

Takie użycie `receive` można traktować jako pokazanie eleganckiego sposobu przekazywania wielu wartości lub też jako pretekst do pokazania sztuczki programistycznej. Taki argument jak `receive`, który jest następną procedurą do wywołania, jest nazywany „kontynuacją”. Przypomnijmy, że kontynuacji używaliśmy również do zaimplementowania struktury sterującej przeszukiwaniem z nawrotami w evaluatorze `amb`, w punkcie 4.3.3.

```
(define (instruction-execution-proc inst)
  (cdr inst))

(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))
```

Tekst instrukcji nie jest wykorzystywany przez nasz symulator, ale dobrze jest go mieć pod ręką na wypadek odpluskowania (zob. ćwiczenie 5.16).

Elementami tablicy etykiet są pary:

```
(define (make-label-entry label-name insts)
  (cons label-name insts))
```

Do wyszukiwania elementów w tablicy służy następująca procedura:

```
(define (lookup-label labels label-name)
  (let ((val (assoc label-name labels)))
    (if val
        (cdr val)
        (error "Niezdefiniowana etykieta -- ASSEMBLE"
label-name))))
```

### Ćwiczenie 5.8

Następujący kod maszyny rejestrowej jest niejednoznaczny, ponieważ etykieta `here` jest w nim zdefiniowana więcej niż raz:

```
start
  (goto (label here))
here
  (assign a (const 3))
  (goto (label there))
here
  (assign a (const 4))
  (goto (label there))
there
```

Jaka będzie zawartość rejestrów a dla opisanego powyżej symulatora, gdy osiągnięta zostanie etykieta `there`? Tak zmodyfikuj procedurę `extract-labels`, aby zgłaszała błąd, gdy ta sama nazwa jest używana na oznaczenie dwóch różnych miejsc.

#### 5.2.3. Generowanie procedur wykonawczych dla instrukcji

Asembler wywołuje procedurę `make-execution-procedure`, aby wygenerować procedurę wykonawczą dla instrukcji. Tak jak procedura `analyze` w evaluatorze z punktu 4.1.7, dokonuje ona rozdziału ze względu na rodzaj instrukcji i generuje odpowiednią procedurę wykonawczą:

```
(define (make-execution-procedure inst labels machine
                                    pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else (error "Nieznaný rodaj instrukcji -- ASSEMBLE"
                     inst))))
```

Dla każdego rodzaju instrukcji w języku opisu maszyn rejestrowych istnieje generator, który konstruuje odpowiednią procedurę wykonawczą. Szczegóły tych procedur wyznaczają zarówno składnię, jak i znaczenie poszczególnych instrukcji w języku opisu maszyn rejestrowych. Stosujemy abstrakcję danych do oddzielenia szczegółów składni wyrażeń maszyny rejestrowej od ogólnego mechanizmu ich wykonywania, tak jak to uczyniliśmy w przypadku evaluatora z punktu 4.1.2, używając procedur składniowych do wydzielania i klasyfikowania części instrukcji.

### Instrukcje assign

Procedura make-assign realizuje instrukcje assign:

```
(define (make-assign inst machine labels operations pc)
  (let ((target
         (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    (let ((value-proc
           (if (operation-exp? value-exp)
               (make-operation-exp
                 value-exp machine labels operations)
               (make-primitive-exp
                 (car value-exp) machine labels))))
      (lambda () ; procedura wykonawcza dla assign
        (set-contents! target (value-proc))
        (advance-pc pc))))))
```

Procedura `make-assign` wydziela z instrukcji `assign` nazwę docelowego rejestru (drugi element instrukcji) oraz wyrażenie określające wartość (pozostała część listy tworzącej instrukcję), używając do tego selektorów

```
(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))

(define (assign-value-exp assign-instruction)
  (cddr assign-instruction))
```

Nazwa rejestru jest sprawdzana za pomocą procedury `get-register`, której wynikiem jest obiekt docelowego rejestru. Wyrażenie określające wartość jest przekazywane do `make-operation-exp`, gdy wartość jest wynikiem operacji, lub do `make-primitive-exp` w przeciwnym razie. Procedury te (pokazane dalej) analizują składnię wyrażenia i tworzą procedurę wykonawczą obliczającą wartość. Jest to procedura bezargumentowa o nazwie `value-proc`, która jest wywoływana w trakcie symulacji w celu uzyskania faktycznej wartości, jaką należy przypisać rejestrowi. Zwróćmy uwagę, że wyszukiwanie nazwy rejestru i analiza składni wyrażenia określającego wartość są wykonywane tylko raz, w trakcie asemblacji, a nie za każdym razem, gdy symulowane jest wykonanie instrukcji. Ta oszczędność pracy stanowi powód, dla którego stosujemy procedury wykonawcze, i odpowiada ona dokładnie oszczędności pracy, którą uzyskaliśmy, oddzielając analizę programu od jego wykonania w evaluatorze z punktu 4.1.7.

Wynikiem `make-assign` jest procedura wykonawcza dla instrukcji `assign`. Gdy procedura ta jest wywoływana (przez procedurę `execute` w modelu maszyny), zapamiętuje ona w rejestrze docelowym wartość uzyskaną jako wynik wykonania procedury `value-proc`. Następnie przesuwa ona `pc` do kolejnej instrukcji, uruchamiając procedurę

```
(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))
```

`Advance-pc` stanowi zwykłe zakończenie wszystkich instrukcji z wyjątkiem `branch` i `goto`.

### Instrukcje `test`, `branch` i `goto`

Procedura `make-test` w podobny sposób realizuje instrukcje `test`. Wydziela ona wyrażenie określające warunek, który ma być zbadany, i generuje dla niego procedurę wykonawczą. W trakcie symulacji jest wywoływana procedura wykonawcza dla danego warunku, a jej wynik jest zapamiętywany w rejestrze `flag`, po czym `pc` jest przesuwany do kolejnej instrukcji:

```
(define (make-test inst machine-labels operations flag pc)
  (let ((condition (test-condition inst)))
```

```

(if (operation-exp? condition)
    (let ((condition-proc
          (make-operation-exp
            condition machine labels operations)))
      (lambda ()
        (set-contents! flag (condition-proc))
        (advance-pc pc)))
    (error "Niepoprawna instrukcja TEST -- ASSEMBLE" inst)))

(define (test-condition test-instruction)
  (cdr test-instruction))

```

Procedura wykonawcza dla instrukcji branch sprawdza zawartość rejestrów flag i albo ustawia rejestr pc na punkt docelowy skoku warunkowego (jeśli warunek jest spełniony), albo tylko przesuwa pc do następnej instrukcji (jeśli warunek nie jest spełniony). Zwróćmy uwagę, że wskazany punkt docelowy skoku w instrukcji branch musi być etykietą, co wymusza procedurę make-branch. Zauważmy również, że miejsce oznaczone etykietą jest ustalane w trakcie asemblacji, a nie za każdym razem, gdy symulowane jest wykonanie instrukcji branch.

```

(define (make-branch inst machine labels flag pc)
  (let ((dest (branch-dest inst)))
    (if (label-exp? dest)
        (let ((insts
              (lookup-label labels (label-exp-label dest))))
          (lambda ()
            (if (get-contents flag)
                (set-contents! pc insts)
                (advance-pc pc)))
        (error "Niepoprawna instrukcja BRANCH -- ASSEMBLE" inst)))

(define (branch-dest branch-instruction)
  (cadr branch-instruction))

```

Instrukcja goto przypomina skok warunkowy z wyjątkiem tego, że punkt docelowy skoku może być określony jako etykietą lub zawartość rejestrów, przy czym żaden warunek nie jest sprawdzany — pc jest zawsze ustawiany na punkt docelowy.

```

(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts
                 (lookup-label labels
                               (label-exp-label dest))))
             (lambda () (set-contents! pc insts))))

```

```

((register-exp? dest)
  (let ((reg
         (get-register machine
                       (register-exp-reg dest))))
    (lambda ()
      (set-contents! pc (get-contents reg))))
  (else (error "Niepoprawna instrukcja GOTO -- ASSEMBLE"
               inst)))))

(define (goto-dest goto-instruction)
  (cadr goto-instruction))

```

### Pozostałe instrukcje

Instrukcje `save` i `restore`, operujące na stosie, wykonują po prostu odpowiednią operację na stosie i wskazanym rejestrze, po czym przesuwają `pc`:

```

(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc)))))

(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc)))))

(define (stack-inst-reg-name stack-instruction)
  (cadr stack-instruction))

```

Ostatni rodzaj instrukcji, realizowany przez procedurę `make-perform`, generuje procedurę wykonawczą dla danej akcji. W czasie symulacji wykonywana jest ta procedura, po czym jest przesuwany `pc`.

```

(define (make-perform inst machine labels operations pc)
  (let ((action (perform-action inst)))
    (if (operation-exp? action)
        (let ((action-proc
               (make-operation-exp
                 action machine labels operations)))
          (lambda ()
            (action-proc)
            (advance-pc pc)))
        (error "Niepoprawna instrukcja PERFORM -- ASSEMBLE" inst)))))

(define (perform-action inst) (cdr inst))

```

### Procedury wykonawcze dla podwyrażeń

Wartości wyrażeń `reg`, `label` lub `const` mogą być przypisywane do rejestrów (`make-assign`) lub też mogą być argumentami operacji (`make-operation-exp` poniżej). Następująca procedura generuje procedury wykonawcze, których wynikami są wartości tych wyrażeń w trakcie symulacji:

```
(define (make-primitive-exp exp machine labels)
  (cond ((constant-exp? exp)
          (let ((c (constant-exp-value exp)))
            (lambda () c)))
        ((label-exp? exp)
          (let ((insts
                  (lookup-label labels
                                (label-exp-label exp))))
            (lambda () insts)))
        ((register-exp? exp)
          (let ((r (get-register machine
                                 (register-exp-reg exp))))
            (lambda () (get-contents r))))
        (else
          (error "Nieznany rodzaj wyrażenia -- ASSEMBLE" exp))))
```

Składnia wyrażeń `reg`, `label` i `const` jest określona przez

```
(define (register-exp? exp) (tagged-list? exp 'reg))
(define (register-exp-reg exp) (cadr exp))
(define (constant-exp? exp) (tagged-list? exp 'const))
(define (constant-exp-value exp) (cadr exp))
(define (label-exp? exp) (tagged-list? exp 'label))
(define (label-exp-label exp) (cadr exp))
```

Instrukcje `assign`, `perform` i `test` mogą powodować wykonanie operacji maszynowej (określonej przez wyrażenie `op`) na argumentach (określonych przez wyrażenia `reg` i `const`). Wynikiem następującej procedury jest procedura wykonawcza dla „wyrażenia operacyjnego” — listy zawierającej operację występującą w instrukcji i wyrażenia określające argumenty:

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations)))
    (aprocs
      (map (lambda (e)
              (make-primitive-exp e machine labels))
        (operation-exp-operands exp))))
```

---

```
(lambda ()
  (apply op (map (lambda (p) (p)) aprocs)))))
```

Składnia wyrażeń operacyjnych jest określona przez

```
(define (operation-exp? exp)
  (and (pair? exp) (tagged-list? (car exp) 'op)))

(define (operation-exp-op operation-exp)
  (cadr (car operation-exp)))

(define (operation-exp-operands operation-exp)
  (cdr operation-exp))
```

Zwróćmy uwagę, że sposób, w jaki traktujemy wyrażenia operacyjne, w dużym stopniu przypomina sposób, w jaki zastosowania procedur są traktowane przez procedurę `analyze-application` z punktu 4.1.7, gdyż w obu przypadkach generujemy procedurę wykonawczą dla każdego argumentu. W trakcie symulacji wywołujemy procedury argumentów i do uzyskanych wyników stosujemy procedurę w języku Scheme, która symuluje daną operację. Procedura symulacyjna jest znajdująca się w tablicy operacji danej maszyny na podstawie nazwy operacji:

```
(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (error "Nieznana operacja -- ASSEMBLE" symbol))))
```

### Ćwiczenie 5.9

Sposób, w jaki powyżej traktowane są operacje, pozwala na stosowanie ich zarówno do etykiet, jak i stałych oraz zawartości rejestrów. Tak zmodyfikuj procedury przetwarzające wyrażenia, aby wymusić spełnienie warunku, że operacje mogą dotyczyć tylko rejestrów i stałych.

### Ćwiczenie 5.10

Zaprojektuj nową składnię instrukcji operujących na rejestrach maszynowych i tak zmodyfikuj symulator, aby używała tej nowej składni. Czy potrafisz zaimplementować nową składnię, zmieniając wyłącznie procedury składniowe przedstawione w tym punkcie?

### Ćwiczenie 5.11

Gdy w punkcie 5.1.4 wprowadziliśmy operacje `save` i `restore`, nie określiliśmy, co się stanie, jeżeli spróbujemy odtworzyć rejestr, który nie został zachowany jako ostatni, jak to ma miejsce w następującym ciągu instrukcji:

```
(save y)
(save x)
	restore y)
```

Istnieje kilka możliwych sensownych znaczeń instrukcji `restore`:

(a) Instrukcja `(restore y)` umieszcza w rejestrze `y` ostatnią zachowaną na stosie wartość bez względu na to, z jakiego pochodzi ona rejestr. W taki właśnie sposób zachowuje się nasz symulator. Pokaż, jak można wykorzystać takie zachowanie i wyeliminować jedną instrukcję z maszyny obliczającej liczby Fibonacciego, przedstawionej w punkcie 5.1.4 (rys. 5.12).

(b) Instrukcja `(restore y)` umieszcza w rejestrze `y` ostatnią zachowaną na stosie wartość, ale tylko jeśli pochodzi ona z rejestru `y`; w przeciwnym razie sygnalizowany jest błąd. Zmodyfikuj symulator, aby działał w ten sposób. Będziesz musiał tak zmienić `save`, aby umieszczało na stosie obok wartości rejestru również jego nazwę.

(c) Instrukcja `(restore y)` umieszcza w rejestrze `y` ostatnią zachowaną wartość pochodząą z rejestrów `y` bez względu na to, jakie inne rejesty były zachowywane po `y` i nie zostały odtworzone. Zmodyfikuj symulator, aby działał w ten sposób. Będziesz musiał z każdym rejestrem związać osobny stos. Operacja `initialize-stack` powinna inicjować stosy wszystkich rejestrów.

### Ćwiczenie 5.12

Symulator może być przydatny przy wyznaczaniu ścieżek danych wymaganych do zaimplementowania maszyny o zadanym sterowniku. Rozszerz asembler tak, aby przechowywał w modelu maszyny następujące informacje:

- listę (bez powtórzeń) wszystkich instrukcji, uporządkowaną ze względu na rodzaj instrukcji (`assign`, `goto` itd.);
- listę (bez powtórzeń) rejestrów używanych do przechowywania punktów wejścia (są to rejesty używane przez instrukcje `goto`);
- listę (bez powtórzeń) rejestrów, które są zachowywane na stosie lub odtwarzane z niego;
- dla każdego rejestrów: listę (bez powtórzeń) źródeł, z których pochodzą przypisywane do niego wartości (np. źródła, z których pochodzą wartości przypisywane do rejestrów `val` w maszynie obliczającej silnię, przedstawionej na rys.5.11, to: `(const 1)` i `((op *) (reg n) (reg val))`).

Rozszerz interfejs komunikatów przekazywanych do maszyny, aby umożliwić dostęp do tych nowych informacji. Aby przetestować powstały analizator, zdefiniuj maszynę obliczającą liczby Fibonacciego (przedstawioną na rys. 5.12) i zbadaj skonstruowane listy.

### Ćwiczenie 5.13

Zmodyfikuj symulator tak, aby wyznaczał rejesty maszyny na podstawie ciągu instrukcji sterownika, a nie wymagał podawania ich listy jako argumentu procedury `make-machine`. Zamiast wstępnie tworzyć rejesty w `make-machine`, możesz tworzyć je po jednym, gdy dany rejestr po raz pierwszy pojawia się w trakcie asemblacji.

#### 5.2.4. Monitorowanie wydajności maszyn

Symulacja może służyć nie tylko do sprawdzania poprawności przedstawianych projektów maszyn, lecz także do mierzenia wydajności maszyn. Możemy na przykład zainstalować w naszym programie symulacyjnym „miernik”, który mierzy liczbę operacji stosowych wykonywanych w trakcie obliczenia. Aby to zrobić, zmodyfikujemy nasz symulowany stos tak, żeby śledził liczbę operacji zachowywania rejestrów na stosie i maksymalną liczbę elementów na stosie, oraz dodamy do interfejsu stosu komunikaty, które powodują wypisanie odpowiednich statystyk, jak jest to pokazane poniżej. Dodamy również do podstawowego modelu maszyny operację, która wypisuje statystyki stosu, inicjując `the-ops` w procedurze `make-new-machine` w następujący sposób:

```
(list (list 'initialize-stack
            (lambda () (stack 'initialize)))
      (list 'print-stack-statistics
            (lambda () (stack 'print-statistics))))
```

Oto nowa wersja make-stack:

```

(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop)
      (if (null? s)
          (error "Pusty stos -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            (set! current-depth (- current-depth 1))
            top)))
    (define (initialize)
      (set! s '())
      (set! number-pushes 0)
      (set! max-depth 0)
      (set! current-depth 0)
      'done)
    (define (print-statistics)
      (newline)
      (display (list 'łączna 'liczba 'włożeń '= number-pushes
                     'maksymalny 'rozmiar '= max-depth)))
    (list 'push 'pop 'initialize 'print-statistics)))
  
```

```
(define (dispatch message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        ((eq? message 'print-statistics)
         (print-statistics))
        (else
         (error "Nieznane polecenie -- STACK" message))))
  dispatch)
```

W ćwiczeniach 5.15–5.19 są opisane inne użyteczne funkcje monitorujące i odpluskwiące, które można dodać do symulatora maszyn rejestrowych.

### Ćwiczenie 5.14

Zmierz liczbę włożeń na stos i maksymalny rozmiar stosu potrzebny do obliczenia  $n!$  dla różnych niewielkich wartości  $n$ , używając maszyny obliczającej silnię przedstawionej na rys. 5.11. Na podstawie uzyskanych wyników podaj wzory określające, w zależności od  $n$ , łączną liczbę włożeń na stos i maksymalny rozmiar stosu potrzebny do obliczenia  $n!$  dla dowolnego  $n > 1$ . Zauważmy, że każda z tych wielkości jest liniową funkcją  $n$  i może być wyznaczona na podstawie wartości w dwóch punktach. W celu wypisania statystyk będziesz musiał dodać do maszyny obliczającej silnię instrukcje inicjujące stos i wypisujące statystyki. Możesz również tak zmodyfikować maszynę, aby wielokrotnie wczytywała wartość  $n$ , obliczała silnię i wypisywała wyniki (jak to zrobiliśmy w przypadku maszyny obliczającej NWD z rys. 5.4), dzięki czemu nie będziesz musiał ciągle wywoływać procedur `get-register-contents`, `set-register-contents!` i `start`.

### Ćwiczenie 5.15

Dodaj do symulatora maszyn rejestrowych *zliczanie instrukcji*. To znaczy spraw, aby model maszyny zliczał liczbę wykonywanych instrukcji. Rozszerz interfejs modelu maszyny o nowy komunikat, który powoduje wypisanie wartości licznika instrukcji i wyzerowanie go.

### Ćwiczenie 5.16

Rozbuduj symulator tak, aby umożliwiał *śledzenie instrukcji*. Oznacza to, że przed wykonaniem każdej instrukcji symulator powinien wypisywać jej tekst. Model maszyny powinien przyjmować komunikaty `trace-on` i `trace-off`, które odpowiednio włączają i wyłączają śledzenie.

### Ćwiczenie 5.17

Rozszerz śledzenie instrukcji z ćwiczenia 5.16 tak, aby przed wypisaniem instrukcji symulator wypisywał wszystkie etykiety występujące w ciągu instrukcji bezpośrednio przed tą instrukcją. Uważaj, aby zrobić to w sposób, który nie będzie się kłócił ze zliczaniem instrukcji (ćwiczenie 5.15). Musisz zapewnić, aby symulator zachowywał konieczne informacje dotyczące etykiet.

### Ćwiczenie 5.18

Zmodyfikuj procedurę `make-register` z punktu 5.2.1 tak, aby można było śledzić rejesty. Rejestry powinny przyjmować komunikaty, które włączają i wyłączają śledzenie. Gdy rejestr jest śledzony, wówczas przypisanie mu wartości powinno spowodować wypisanie nazwy rejestru oraz jego starej i nowej zawartości. Rozszerz interfejs modelu maszyny tak, aby można było włączać i wyłączać śledzenie określonych rejestrów maszyny.

### Ćwiczenie 5.19

Liz P. Haker chciałaby, aby symulator umożliwiał wstawianie do programu *punktów kontrolnych* (ang. *break points*), pomagając jej w odpluskowaniu projektów maszyn. Został wynajęty do zainstalowania tego dla niej. Chciałaby ona móc określać miejsca w ciągu instrukcji sterownika, w których symulator zatrzymywałby się i pozwalał jej badać stan maszyny. Masz zaimplementować procedurę

```
(set-breakpoint <maszyna> <etykieta> <n>)
```

która wstawia punkt kontrolny tuż przed *n*-tą instrukcją po danej etykiecie. Na przykład

```
(set-breakpoint gcd-machine 'test-b 4)
```

wstawia punkt kontrolny w gcd-machine tuż przed przypisaniem do rejestrów a. Gdy symulator dochodzi do punktu kontrolnego, powinien wypisać etykietę i przesunięcie punktu kontrolnego oraz przerwać wykonywanie instrukcji. Liz może wówczas manipulować stanem symulowanej maszyny za pomocą operacji `get-register-contents` i `set-register-contents!`. Potem powinna ona móc kontynuować wykonanie, wprowadzając polecenie

```
(proceed-machine <maszyna>)
```

Powinna również móc usunąć określony punkt kontrolny za pomocą

```
(cancel-breakpoint <maszyna> <etykieta> <n>)
```

lub też usunąć wszystkie punkty kontrolne za pomocą

```
(cancel-all-breakpoints <maszyna>)
```

## 5.3. Przydzielanie pamięci i odśmiecanie

W podrozdziale 5.4 pokażemy, jak można zaimplementować evaluator języka Scheme jako maszynę rejestrową. Dla uproszczenia rozważań zakładamy, że nasze maszyny rejestrowe mogą być wyposażone w *pamięć struktur listowych* (ang. *list-structured memory*), w której podstawowe operacje na listowych strukturach danych są operacjami pierwotnymi. Postulowanie istnienia takiej pamięci jest użyteczną formą abstrakcji, gdy skupiamy się na mechanizmach sterowania w interpreterze języka Scheme, nie oddaje jednak wiernie

faktycznych operacji pierwotnych we współczesnych komputerach. Chcąc uzyskać pełniejszy obraz tego, jak działają systemy lispowe, musimy zbadać, jak struktury listowe mogą być reprezentowane w sposób zgodny z konstrukcją konwencjonalnych pamięci komputerowych.

Przy implementowaniu struktur listowych należy uwzględnić dwie kwestie. Pierwsza jest całkowicie związana ze sposobem reprezentacji: jak reprezentować „pułapkowo-wskaźnikową” strukturę lispowych par, mając do dyspozycji możliwości zapamiętywania i adresowania takie jak w typowych pamięciach komputerowych. Druga kwestia dotyczy zarządzania pamięcią w miarę postępowania obliczeń. Działanie systemu lispowego w istotny sposób zależy od możliwości ciągłego tworzenia nowych obiektów danych. Dotyczy to zarówno obiektów jawnie tworzonych przez interpretowane procedury, jak i struktur tworzonych przez sam interpreter, takich jak środowiska czy listy argumentów. Mimo że ciągłe tworzenie nowych obiektów danych nie stanowi problemu w przypadku komputera wyposażonego w nieskończoną pamięć o błyskawicznym dostępie, jednak w handlu są dostępne jedynie skończone pamięci komputerowe (co za szkoda!). Dlatego też systemy lispowe udostępniają *mechanizm automatycznego przydzielania pamięci* (ang. *automatic storage allocation facility*), aby sprawić wrażenie, że dostępna pamięć jest nieskończona. Gdy obiekt danych nie jest już potrzebny, zajmowana przez niego pamięć jest automatycznie zwalniana i wykorzystywana do tworzenia nowych obiektów danych. Istnieje wiele metod umożliwiających takie automatyczne przydzielanie pamięci. Metoda, którą omówimy w niniejszym podrozdziale, jest nazywana *odśmiecaniem* (ang. *garbage collection*).

### 5.3.1. Pamięć jako wektory

Tradycyjną pamięć komputerową możemy sobie wyobrazić jako tablicę komórek, z których każda może zawierać porcję informacji. Każda komórka ma jednoznaczną nazwę, zwaną jej *adresem* lub *lokacją*. Typowe systemy pamięci udostępniają dwie operacje pierwotne: jedną do odczytywania danych przechowywanych pod określonym adresem i jedną do zapisywania nowych danych do komórki o zadanym adresie. Adresy pamięci można zwiększać, realizując sekwencyjny dostęp do zestawów komórek. Ogólnie rzecz biorąc, wiele ważnych operacji na danych wymaga, aby adresy pamięci mogły być traktowane jak dane, które można przechowywać w pamięci i które można przetwarzać w rejestrach maszynowych. Reprezentacja struktur listowych jest jednym z zastosowań takiej *arytmetyki adresów*.

Do modelowania pamięci komputerowej użyjemy nowego rodzaju struktury danych nazywanej *wektorem*. Z abstrakcyjnego punktu widzenia wektor jest złożonym obiektem danych, którego poszczególne elementy są dostępne po-

przez całkowite indeksy, w czasie niezależnym od wartości indeksu<sup>5</sup>. Opisując operacje na pamięci, będziemy używać dwóch procedur pierwotnych języka Scheme operujących na wektorach:

- (`(vector-ref <wektor> <n>)`) daje w wyniku  $n$ -ty element wektora;
- (`(vector-set! <wektor> <n> <wartość>)`) ustawia wartość  $n$ -tego elementu danego wektora.

Jeśli na przykład `v` jest wektorem, to wywołanie (`(vector-ref v 5)`) daje w wyniku wartość piątego elementu wektora `v`, a wywołanie (`(vector-set! v 5 7)`) zmienia wartość piątego elementu wektora `v` na `7`<sup>6</sup>. W przypadku pamięci komputerowej taki dostęp może być zaimplementowany przy użyciu arytmetyki adresów do łączenia *adresu bazowego*, określającego adres początku wektora w pamięci, z *indeksem*, który określa przesunięcie konkretnego elementu wektora względem początku.

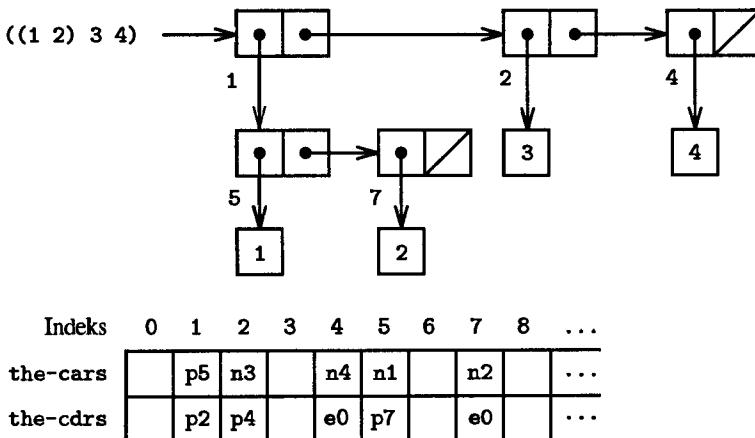
### Reprezentowanie danych lispowych

Możemy użyć wektorów do zaimplementowania podstawowych struktur par potrzebnych do zrealizowania pamięci struktur listowych. Wyobraźmy sobie, że pamięć komputera jest podzielona na dwa wektory: `the-cars` i `the-cdrs`. Struktury listowe będziemy reprezentować w następujący sposób: Wskaźnik do pary to indeks w obydwu wektorach. Car pary to element wektora `the-cars` o danym indeksie, a cdr pary to element wektora `the-cdrs` o danym indeksie. Musimy również reprezentować obiekty inne niż pary (takie jak liczby lub symbole) i potrzebny nam będzie sposób odróżniania jednego rodzaju danych od innych. Istnieje wiele sposobów realizacji tego, ale wszystkie one sprawadzają się do użycia wskaźników ze znacznikiem typu, tzn. do rozszerzenia pojęcia „wskaźnika”, aby zawierał on informacje o typie danych<sup>7</sup>. Na podstawie typu danych system odróżnia wskaźnik do pary (który składa się z typu danych „pary” oraz indeksu wektorów pamięci) od wskaźników do innych rodzajów danych (które składają się z innych typów danych i czegoś, co jest używane do reprezentowania danych tego typu). Dwa obiekty danych są uważa

<sup>5</sup> Moglibyśmy reprezentować pamięć jako listy elementów. Wówczas jednak czas dostępu nie byłby niezależny od wartości indeksu, gdyż dostęp do  $n$ -tego elementu listy wymaga wykonania  $n - 1$  operacji `cdr`.

<sup>6</sup> Dla pełności opisu powinniśmy określić operację `make-vector`, która tworzy wektory. Jednak w tym zastosowaniu będziemy używać wektorów jedynie do modelowania ustalonych porcji pamięci komputerowej.

<sup>7</sup> Jest to dokładnie ta sama koncepcja „danych ze znacznikami”, którą wprowadziliśmy w rozdziale 2 w celu realizacji operacji ogólnych. Tutaj jednak typy danych są wprowadzane na poziomie maszyny pierwotnej, a nie tworzone za pomocą list.



Rys. 5.14. Reprezentacja listy  $((1\ 2)\ 3\ 4)$  w postaci diagramu pudełkowo-wskaźnikowego oraz wektorów pamięci

żane za tożsame (`eq?`), jeśli ich wskaźniki są identyczne<sup>8</sup>. Na rysunku 5.14 jest przedstawione użycie tej metody do reprezentacji listy  $((1\ 2)\ 3\ 4)$ , której diagram pudełkowo-wskaźnikowy jest również pokazany. Do oznaczenia informacji o typach danych używamy jednoliterowych przedrostków. Tak więc wskaźnik (ang. *pointer*) do pary o indeksie 5 jest oznaczony jako `p5`, lista pusta (ang. *empty*) jest oznaczona jako wskaźnik `e0`, a wskaźnik do liczby (ang. *number*) 4 jest oznaczony jako `n4`. Na diagramie pudełkowo-wskaźnikowym, przy dolnym lewym rogu pudełka pary jest zaznaczony indeks określający, gdzie są pamiętane car i cdr tej pary. Puste komórki w `the-cars` i `the-cdrs` mogą zawierać części innych struktur listowych (nie interesującym nas tutaj).

Wskaźnik do liczby, taki jak `n4`, mógłby się składać z typu wskazującego dane liczbowe oraz faktycznej reprezentacji liczby 4<sup>9</sup>. Chcąc radzić sobie z liczbami, które są zbyt duże, aby można było reprezentować je w ustalonej

<sup>8</sup> Informacja o typie może być zakodowana na wiele możliwych sposobów w zależności od szczegółów maszyny, na której ma być zaimplementowany system lispowy. Od tego, jak sprytnie zostanie to zrealizowane, w istotny sposób będzie zależeć efektywność wykonania programów lispowych. Trudno jednak sformułować ogólne zasady wskazujące dobrą wybór. Najprostszym sposobem zaimplementowania wskaźników polega na przeznaczeniu ustalonego zestawu bitów w każdym wskaźniku na *pole typu*, w którym zakodowany jest typ danych. Wśród istotnych kwestii, które należy rozważyć, projektując taką reprezentację, należy wymienić następujące: Ile bitów typu potrzeba? Jak duże muszą być indeksy wektorów? Jak efektywnie można operować za pomocą pierwotnych operacji maszynowych na polach typu wskaźników? O maszynach, które zawierają specjalne układy sprzętowe służące do efektywnego przetwarzania pól typu, mówi się, że mają *architekturę znacznikową*.

<sup>9</sup> Taka decyzja dotycząca reprezentacji liczb określa, czy predykat `eq?`, który sprawdza równość wskaźników, może być używany do porównywania liczb. Jeśli wskaźnik zawiera samą liczbę, to równe liczby będą miały ten sam wskaźnik. Jeśli jednak wskaźnik zawiera indeks komórki, w której pamiętana jest liczba, to będziemy mogli polegać na tym, że równe liczby mają równe

ilości pamięci przeznaczonej na jeden wskaźnik, możemy używać odrębnego typu danych `bignum`, w przypadku którego wskaźnik wyznacza listę, na której są pamiętane części tej liczby<sup>10</sup>.

Symbol może być reprezentowany jako wskaźnik ze znacznikiem typu, wyznaczający ciąg znaków, które składają się na pisemną reprezentację symbolu. Taki ciąg jest tworzony przez lispową procedurę wczytującą, gdy napotyka ona dany ciąg znaków po raz pierwszy na wejściu. Ponieważ chcemy, aby dwa wystąpienia symbolu były uznawane przez `eq?` za „ten sam” symbol, oraz chcemy, aby `eq?` było prostym porównywaniem wskaźników, musimy zagwarantować, że gdy procedura wczytująca napotka ten sam ciąg znaków po raz kolejny, wtedy użycie tego samego wskaźnika (do tego samego ciągu znaków) na oznaczenie obydwóch wystąpień symbolu. Aby to zapewnić, procedura wczytująca utrzymuje tablicę wszystkich symboli, jakie kiedykolwiek napotkała, tradycyjnie nazywaną `obarray`. Gdy procedura wczytująca napotyka napis i ma zbudować symbol, sprawdza w `obarray`, czy już spotkała taki napis. Jeśli nie, to tworzy z wczytanych znaków nowy symbol (wskaźnik ze znacznikiem typu do nowego ciągu znaków) i wprowadza ten wskaźnik do `obarray`. Jeśli natomiast procedura wczytująca widziała już taki napis wcześniej, to przekazuje jako symbol wskaźnik zapamiętany w `obarray`. Taki proces zastępowania napisów jednoznacznymi wskaźnikami jest nazywany *katalogowaniem* symboli.

### Implementowanie operacji pierwotnych na listach

Przy przedstawionym powyżej schemacie reprezentacji możemy zastąpić każdą „pierwotną” operację na listach maszyny rejestrowej jedną lub więcej operacjami pierwotnymi na wektorach. Będziemy używać dwóch rejestrów `the-cars` i `the-cdrs` na oznaczenie wektorów pamięci oraz będziemy zakładać, że `vector-ref` i `vector-set!` są dostępne jako operacje pierwotne. Zakładamy również, że operacje arytmetyczne na wskaźnikach (takie jak zwiększanie wskaźnika, użycie wskaźnika do pary do indeksowania wektora lub dodawanie dwóch liczb) korzystają tylko z części indeksowej wskaźnika ze znacznikiem typu.

Mogemy na przykład sprawić, aby maszyna rejestrowa udostępniała instrukcje

```
(assign (r1) (op car) (reg (r2)))
(assign (r1) (op cdr) (reg (r2)))
```

wskaźniki tylko wtedy, kiedy będziemy uważać, aby nigdy nie przechowywać tej samej liczby w więcej niż jednej komórce.

<sup>10</sup> To tak, jak byśmy zapisali liczbę jako ciąg cyfr, przy czym każda „cyfra” jest liczbą od 0 do największej liczby, jaką można zapamiętać w jednym wskaźniku.

jeśli zaimplementujemy je, odpowiednio, jako

```
(assign ⟨r1⟩ (op vector-ref) (reg the-cars) (reg ⟨r2⟩))
(assign ⟨r1⟩ (op vector-ref) (reg the-cdrs) (reg ⟨r2⟩))
```

Instrukcje

```
(perform (op set-car!) (reg ⟨r1⟩) (reg ⟨r2⟩))
(perform (op set-cdr!) (reg ⟨r1⟩) (reg ⟨r2⟩))
```

są zaimplementowane jako

```
(perform
  (op vector-set!) (reg the-cars) (reg ⟨r1⟩) (reg ⟨r2⟩))
(perform
  (op vector-set!) (reg the-cdrs) (reg ⟨r1⟩) (reg ⟨r2⟩))
```

Wykonanie operacji `cons` polega na przydzieleniu nieużywanego indeksu  $i$  i zapamiętaniu argumentów `cons` w wektorach `the-cars` i `the-cdrs` pod odpowiednim indeksem. Przyjmujemy tutaj, że istnieje specjalny rejestr `free`, który zawsze zawiera wskaźnik do pary wskazującej na pierwszy wolny indeks, oraz że możemy uzyskać wskaźnik do następnej wolnej pozycji, zwiększając część indeksową tego wskaźnika<sup>11</sup>. Na przykład instrukcja

```
(assign ⟨r1⟩ (op cons) (reg ⟨r2⟩) (reg ⟨r3⟩))
```

jest zaimplementowana w postaci następującego ciągu operacji wektorowych<sup>12</sup>:

```
(perform
  (op vector-set!) (reg the-cars) (reg free) (reg ⟨r2⟩))
(perform
  (op vector-set!) (reg the-cdrs) (reg free) (reg ⟨r3⟩))
(assign ⟨r1⟩ (reg free))
(assign free (op +) (reg free) (const 1))
```

Predykat `eq?`:

```
(op eq?) (reg ⟨r1⟩) (reg ⟨r2⟩))
```

<sup>11</sup> Istnieją inne sposoby wyznaczania wolnych obszarów pamięci. Moglibyśmy na przykład połączyć wszystkie wolne pary w *listę wolnych indeksów*. W naszym przypadku wolne pozycje następują po sobie (i dlatego możemy je wyznaczać, zwiększając wskaźnik `free`), gdyż będziemy używać odśmiecania ze scalaniem bloków, opisanego w punkcie 5.3.2.

<sup>12</sup> Jest to zasadniczo implementacja operacji `cons` za pomocą `set-car!` i `set-cdr!`, jak to zostało opisane w punkcie 3.3.1. Używana tam operacja `get-new-pair` jest tutaj realizowana za pomocą wskaźnika `free`.

po prostu sprawdza równość wszystkich pól w rejestrach, a takie predykaty, jak `pair?`, `null?`, `symbol?` czy `number?`, muszą tylko sprawdzać pole typu.

### Implementowanie stosu

Mimo że nasze maszyny rejestrowe korzystają ze stosów, nie wymaga to żadnych specjalnych zabiegów, gdyż stos można modelować za pomocą listy. Stos może być listą zapamiętanych wartości, wskazywaną przez specjalny rejestr `the-stack`. Tak więc operacja (`save <r>`) może być zaimplementowana jako

```
(assign the-stack (op cons) (reg <r>) (reg the-stack))
```

Podobnie, operacja (`restore <r>`) może być zaimplementowana jako

```
(assign <r> (op car) (reg the-stack))
(assign the-stack (op cdr) (reg the-stack))
```

a operacja (`perform (op initialize-stack)`) może być zaimplementowana jako

```
(assign the-stack (const ()))
```

Operacje te można dalej rozwinąć, używając przedstawionych wcześniej operacji wektorowych. Jednak w tradycyjnych architekturach komputerów zwykle opłaca się przydzielać pamięć stosu w postaci osobnego wektora. Wówczas wkładanie i zdejmowanie elementów ze stosu może być zrealizowane za pomocą zwiększania i zmniejszania indeksu takiego wektora.

### Ćwiczenie 5.20

Narysuj pudełkowo-wskaźnikową i wektorową (taką jak na rys. 5.14) reprezentację struktury listowej powstającej w wyniku wykonania

```
(define x (cons 1 2))
(define y (list x x))
```

dla wskaźnika `free` początkowo równego `p1`. Jaka jest końcowa wartość `free`? Jakie wskaźniki reprezentują wartości `x` i `y`?

### Ćwiczenie 5.21

Zaimplementuj następujące procedury w postaci maszyn rejestrowych. Załóż, że operacje listowe na pamięci są dostępne jako operacje pierwotne maszyny.

(a) Rekurencyjna procedura `count-leaves`:

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                  (count-leaves (cdr tree))))))
```

(b) Rekurencyjna procedura `count-leaves` z jawnym licznikiem:

```
(define (count-leaves tree)
  (define (count-iter tree n)
    (cond ((null? tree) n)
          ((not (pair? tree)) (+ n 1))
          (else (count-iter (cdr tree)
                            (count-iter (car tree) n))))))
  (count-iter tree 0))
```

### Ćwiczenie 5.22

W ćwiczeniu 3.12, w punkcie 3.3.1, przedstawiliśmy procedurę `append`, która tworzy nową listę będącą sklejeniem dwóch danych list, oraz procedurę `append!`, która skleja dwie dane listy razem. Zaprojektuj maszyny rejestrowe implementujące te dwie procedury. Założź, że operacje listowe na pamięci są dostępne jako operacje pierwotne.

#### 5.3.2. Podtrzymywanie złudzenia pamięci nieskończonej

Sposób reprezentacji danych naszkicowany w punkcie 5.3.1 rozwiązuje problem implementowania struktur listowych, pod warunkiem że dostępna jest nieskończona ilość pamięci. W prawdziwym komputerze w końcu zabraknie nam pamięci do konstruowania nowych par<sup>13</sup>. Jednak większość par powstających w typowych obliczeniach służy tylko do przechowywania wyników pośrednich. Po wykorzystaniu wyników pary te nie są już dłużej potrzebne — stanowią śmieci. Na przykład w trakcie obliczania

```
(accumulate + 0 (filter odd? (enumerate-interval 0 n)))
```

powstają dwie listy: lista elementów przedziału i lista odfiltrowanych elementów. Po skumulowaniu elementów listy te nie są już potrzebne i zajmowana przez nie pamięć mogłaby zostać zwrócona. Gdybyśmy mogli zorganizować okresowe „zbieranie śmieci” i gdyby okazało się, że w ten sposób możemy utylizować zużytą pamięć w tym samym tempie, w jakim tworzone są nowe pary, to zachowalibyśmy złudzenie, że dostępna pamięć jest nieskończona.

Chcąc utylizować pary, musimy być w stanie określić, które z alokowanych par nie są już potrzebne (w tym sensie, że ich zawartość nie ma już wpływu na dalsze obliczenia). Metoda, którą zbadamy w tym celu, jest znana jako *odśmiecanie*. Odśmiecane opiera się na tym, że w dowolnym momencie w trakcie

<sup>13</sup> Może to w końcu przestać być prawdą, ponieważ pamięci mogą się stać na tyle duże, że nie będzie możliwe zużycie całej pamięci w czasie istnienia komputera. Na przykład, rok ma około  $3 \times 10^{13}$  mikrosekund, tak więc gdybyśmy wykonywali operację `cons` raz na mikrosekundę, to potrzebowalibyśmy około  $10^{15}$  komórek pamięci, żeby zbudować maszynę, która mogłaby działać przez 30 lat, zanim zabrakłoby jej pamięci. Z jednej strony, tak duża pamięć wydaje się według dzisiejszych standardów absurdalnie wielka, nie jest jednak fizycznie niemożliwa. Z drugiej strony, produkowane procesory są coraz szybsze, a przyszłe komputery mogą być wyposażone w dużą liczbę procesorów operujących wspólnie na wspólnej pamięci, by móc zatem będzie możliwe dużo szybsze zużywanie pamięci, niż to postulowaliśmy.

interpretowania programu w Lispie jedyne obiekty, które mogą wpływać na dalszy przebieg obliczenia, to te, do których można dojść, wykonując w pewnej kolejności operacje `car` i `cdr`, począwszy od wskaźników przechowywanych w danej chwili w rejestrach maszyny<sup>14</sup>. Każda komórka pamięci, która nie jest osiągalna w ten sposób, może być ponownie wprowadzona do użycia.

Odśmiecanie może być zrealizowane na wiele sposobów. Metoda, którą tutaj zbadamy, jest nazywana *odśmiecaniem przez kopiowanie* (ang. *stop-and-copy*). Podstawowy pomysł polega na podzieleniu pamięci na dwie części: „pamięć roboczą” i „wolną pamięć”. Gdy za pomocą `cons` tworzymy parę, przydzielamy jej pamięć z obszaru roboczego. Gdy pamięć robocza się zapłni, odśmiecamy ją, odnajdując w pamięci roboczej wszystkie przydatne pary i kopiując je do kolejnych pozycji w wolnej pamięci. (Przydatne pary odnajdujemy, przechodząc wszystkie wskaźniki `car` i `cdr`, począwszy od rejestrów maszynowych). Ponieważ nie kopiujemy śmieci, przypuszczalnie zostanie trochę wolnej pamięci, którą można przydzieleć nowym param. Ponadto nie potrzebujemy już niczego z pamięci roboczej, gdyż wszystkie przydatne pary zostały z niej skopiowane. Tak więc jeśli zamienimy rolami pamięć roboczą i wolną pamięć, możemy kontynuować obliczenia — nowe pary będą umieszczane w nowej pamięci roboczej (czyli starej wolnej pamięci). Gdy ta się zapłni, możemy przepisać przydatne pary do nowej wolnej pamięci (czyli starej pamięci roboczej)<sup>15</sup>.

<sup>14</sup> Zakładamy tutaj, że stos jest reprezentowany w postaci listy, tak jak to opisaliśmy w punkcie 5.3.1, a elementy przechowywane na stosie są dostępne poprzez wskaźnik przechowywany w rejestrze stosu.

<sup>15</sup> Technika ta została wynaleziona i po raz pierwszy zaimplementowana przez Minsky'ego, jako część implementacji Lispu na komputer PDP-1 opracowanej w MIT Research Laboratory of Electronics. Była ona dalej rozwijana przez Fenichela i Yochelsona [27] na potrzeby implementacji Lispu w systemie Multics z podziałem czasu. Później Baker [6] opracował wersję tej metody działającą w „czasie rzeczywistym”, która nie wymagała zatrzymywania obliczeń na czas odśmiecania. Pomysł Bakera został rozszerzony przez Hewitta, Liebermana i Moona (zob. [67]) w celu wykorzystania faktu, że niektóre struktury są mniej, a niektóre bardziej trwałe.

Alternatywna, powszechnie stosowana technika odśmiecania to metoda *odśmiecania przez zaznaczanie i zamiatanie* (ang. *mark-sweep*). Polega ona na przeglądaniu całej struktury dostępnej poprzez rejesty maszynowe i zaznaczaniu wszystkich odwiedzanych par. Następnie cała pamięć jest przeglądana i wszystkie pozycje, które nie są zaznaczone, są „wymiatane” jako śmieci, a zajmowana przez nie pamięć jest ponownie dostępna. Pełne omówienie metody odśmiecania przez zaznaczanie i zamiatanie można znaleźć w książce Allen [2].

Algorytm Minsky'ego-Fenichela-Yochelsona ma największe znaczenie w systemach z dużą pamięcią, gdyż bada on tylko użyteczną część pamięci. W odróżnieniu od tego w odśmiecaniu przez zaznaczanie i zamiatanie, w fazie zamiatania trzeba przejrzeć całą pamięć. Drugą zaletą odśmiecania przez kopiowanie jest to, że jest to technika odśmiecania ze *scalaniem*. Oznacza to, że po zakończeniu odśmiecania wszystkie przydatne pary zajmują spójny obszar pamięci, bez dziur po śmieciach. Może mieć to niezwykłe istotny wpływ na wydajność w przypadku maszyn z pamięcią wirtualną, w których dostęp do adresów porozrzucanych w pamięci może powodować dodatkowe operacje wymiany stron.

## Implementacja odśmiecania przez kopiowanie

Użyjemy teraz naszego języka opisu maszyn rejestrowych do bardziej szczegółowo opisania algorytmu odśmiecania przez kopiowanie. Zakładamy, że istnieje rejestr o nazwie `root` (korzeń, źródło), zawierający wskaźnik do struktury, która w końcu wskazuje na wszystkie dostępne dane. Można to osiągnąć, przechowując zawartość wszystkich rejestrów maszynowych na liście wskazywanej przez `root`, utworzonej tuż przed rozpoczęciem odśmiecania<sup>16</sup>. Zakładamy również, że oprócz bieżącej pamięci roboczej dostępna jest wolna pamięć, do której można przepisywać przydatne dane. Bieżąca pamięć robocza składa się z wektorów, których adresy bazowe są pamiętane w rejestrach `the-cars` i `the-cdrs`, a wolna pamięć jest pamiętana w rejestrach `new-cars` i `new-cdrs`.

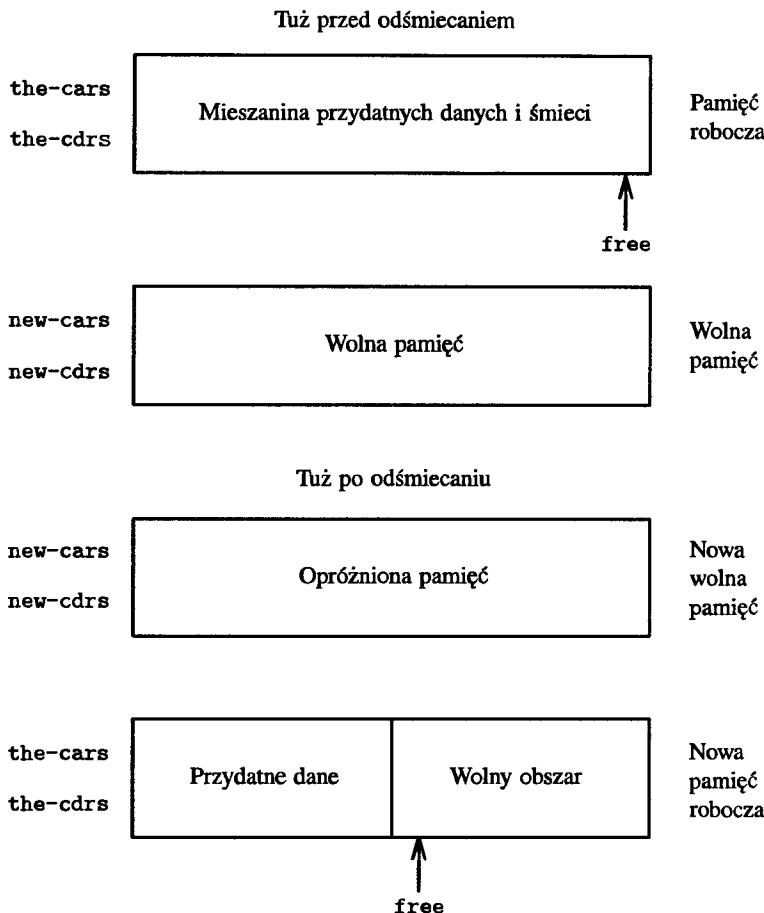
Odśmiecanie jest uruchamiane, gdy zostaną wyczerpane wolne komórki w bieżącej pamięci roboczej, tzn. gdy operacja `cons` próbuje przesunąć wskaźnik `free` poza koniec wektora pamięci. Gdy odśmiecanie zostaje zakończone, wskaźnik `root` wskazuje na nową pamięć, wszystkie obiekty osiągalne poprzez `root` są przeniesione do nowej pamięci, a wskaźnik `free` wskazuje na pierwszą pozycję w nowej pamięci, gdzie może być umieszczona nowa para. Ponadto pamięć robocza i nowa pamięć zamieniają się rolami — nowe pary będą tworzone w nowej pamięci, począwszy od miejsca wskazywanego przez `free`, a (poprzednia) pamięć robocza będzie dostępna przy kolejnym odśmiecaniu jako nowa pamięć. Na rysunku 5.15 jest przedstawiony stan pamięci tuż przed i tuż po odśmiecaniu.

Stan procesu odśmiecania jest śledzony za pomocą dwóch wskaźników: `free` i `scan`. Początkowo wskazują one na początek nowej pamięci. Algorytm rozpoczyna działanie od przeniesienia pary wskazywanej przez `root` na początek nowej pamięci. Para ta jest kopowana, wskaźnik `root` jest ustawiany na jej nowe położenie, a wskaźnik `free` jest zwiększały. Dodatkowo zaznacza się w starej pozycji pary, że jej zawartość została przeniesiona. To zaznaczanie odbywa się w następujący sposób: W miejsce `car` umieszczamy specjalny znacznik sygnalizujący, że dany obiekt został już przeniesiony. (Taki obiekt jest tradycyjnie nazywany *złamane sercem*)<sup>17</sup>. W miejsce `cdr` umieszczamy *adres przekierowania*, który wskazuje miejsce, w które obiekt został przeniesiony.

Po przeniesieniu korzenia (`root`) struktury danych odśmiecanie rozpoczyna swój podstawowy cykl. W każdym kroku algorytmu wskaźnik `scan` (po-

<sup>16</sup> Lista ta nie obejmuje rejestrów używanych przez system przydziału pamięci: `root`, `the-cars`, `the-cdrs` i innych rejestrów, które zostaną wprowadzone w niniejszym punkcie.

<sup>17</sup> Określenie *złamane serce* (ang. *broken heart*) zostało ukute przez Davida Cresseyea, autora programu odśmiecającego dla języka MDL, dialekta Lispu opracowanego w MIT we wczesnych latach siedemdziesiątych XX w.



Rys. 5.15. Zmiana organizacji pamięci na skutek odśmiecania

czątkowo wskazujący na przeniesiony korzeń) wskazuje na parę, która została przeniesiona do nowej pamięci, ale której car i cdr nadal wskazują na obiekty w starej pamięci. Każdy z tych obiektów jest przenoszony, a wskaźnik scan jest zwiększany. Chcąc przenieść obiekt (np. obiekt wskazywany przez car pary, którą przeglądamy), sprawdzamy, czy nie został on już przeniesiony (co określa znacznik złamanego serca umieszczony w miejscu car obiektu). Jeśli obiekt nie został jeszcze przeniesiony, kopujemy go w miejsce wskazywane przez free, aktualniemy wskaźnik free, umieszczamy znacznik złamane- go serca w miejscu starego położenia obiektu i aktualniemy wskaźnik do obiektu (w tym przykładzie car przeglądanej pary) tak, aby wskazywał jego nowe położenie. Jeśli obiekt został już przeniesiony, to w miejsce wskaźnika w przeglądanej parze wstawiamy adres przekierowania (zapisany w miejscu cdr złamanej serca). W końcu wszystkie dostępne obiekty zostają przenie-

sione i przejrzone. Wówczas wskaźnik `scan` wyprzedza `free` i proces odśmiecania się zatrzymuje.

Możemy określić algorytm kopiowania w formie ciągu instrukcji maszyny rejestrowej. Podstawowy krok polegający na przeniesieniu obiektu jest wykonywany przez podprogram o nazwie `relocate-old-result-in-new`. Podprogram ten pobiera argument (wskaźnik do przenoszonego obiektu) z rejestru o nazwie `old` (stary). Przenosi on wyznaczony obiekt (zwiększając w tym samym czasie wskaźnik `free`), umieszcza wskaźnik do przeniesionego obiektu w rejestrze o nazwie `new` (nowy) i powraca, skacząc do miejsca wywołania zapamiętanego w rejestrze `relocate-continue`. Rozpoczynając odśmiecanie, po zainicjowaniu `free` i `scan`, wywołujemy ten podprogram w celu przeniesienia obiektu wskazywanego przez `root`. Gdy obiekt ten zostanie już przeniesiony, zapamiętujemy jego nowy adres w rejestrze `root` i wchodzimy do głównej pętli odśmiecania.

```
begin-garbage-collection
  (assign free (const 0))
  (assign scan (const 0))
  (assign old (reg root))
  (assign relocate-continue (label reassign-root))
  (goto (label relocate-old-result-in-new))

reassign-root
  (assign root (reg new))
  (goto (label gc-loop))
```

W głównej pętli odśmiecania musimy określić, czy są jeszcze jakieś obiekty do przejrzenia. Robimy to, sprawdzając, czy wskaźnik `scan` pokrywa się ze wskaźnikiem `free`. Jeśli wskaźniki te są równe, to wszystkie dostępne obiekty zostały przeniesione i skaczemy do `gc-flip`, gdzie wszystko jest doprowadzane do porządku, tak że możemy kontynuować przerwane obliczenie. Jeśli nadal są pary do przejrzenia, wywołujemy podprogram przenoszący obiekty w celu przeniesienia car kolejnej pary (umieszczając jej `car` w rejestrze `old`). Rejestr `relocate-continue` jest tak ustawiony, aby po powrocie z podprogramu wskaźnik `car` został uaktualniony.

```
gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip))
  (assign old (op vector-ref) (reg new-cars) (reg scan))
  (assign relocate-continue (label update-car))
  (goto (label relocate-old-result-in-new))
```

W miejscu oznaczonym etykietą `update-car` modyfikujemy `car` przeglądanej pary, a następnie przenosimy obiekt wskazywany przez `cdr` pary. Po

przeniesieniu obiektu wracamy do miejsca oznaczonego etykietą update-cdr. Uaktualnienie cdr kończy przeglądanie pary, po czym kontynuujemy wykonywanie pętli głównej.

```
update-car
  (perform
    (op vector-set!) (reg new-cars) (reg scan) (reg new))
    (assign old (op vector-ref) (reg new-cdrs) (reg scan))
    (assign relocate-continue (label update-cdr))
    (goto (label relocate-old-result-in-new))

update-cdr
  (perform
    (op vector-set!) (reg new-cdrs) (reg scan) (reg new))
    (assign scan (op +) (reg scan) (const 1))
    (goto (label gc-loop))
```

Podprogram relocate-old-result-in-new przenosi obiekty w następujący sposób: Jeśli obiekt, który należy przenieść (wskazywany przez old), nie jest parą, to (w rejestrze new) przekazujemy ten sam niezmieniony wskaźnik do obiektu. (Możemy na przykład przeglądać parę, której car jest liczbą 4. Jeśli reprezentujemy car jako n4, jak to było opisane w punkcie 5.3.1, to chcemy, aby „przeniesiony” wskaźnik car nadal był równy n4). W przeciwnym razie musimy dokonać przeniesienia. Jeśli w miejscu car pary przeznaczonej do przeniesienia znajduje się znacznik złamane serca, to para ta została już w rzeczywistości przeniesiona, odczytujemy więc (z cdr złamane serca) adres przekierowania i przekazujemy go w rejestrze new. Jeśli natomiast wskaźnik w rejestrze old wskazuje na parę, która nie została jeszcze przeniesiona, to przepisujemy ją do pierwszej wolnej komórki w nowej pamięci (wskazywanej przez free) i tworzymy złamane serce, zapisując znacznik złamane serca, i adres przekierowania w miejscu starego położenia obiektu. Podprogram relocate-old-result-in-new używa rejestrów oldcr do przechowywania car lub cdr obiektu wskazywanego przez old<sup>18</sup>.

```
relocate-old-result-in-new
  (test (op pointer-to-pair?) (reg old))
  (branch (label pair))
  (assign new (reg old))
  (goto (reg relocate-continue))
```

<sup>18</sup> Przy odśmiecaniu korzysta się z niskopoziomowego predykatu **pointer-to-pair?** zamiast z predykatu **pair?** odnoszącego się do struktur listowych, ponieważ w prawdziwym systemie różne rzeczy mogą być traktowane na potrzeby odśmiecania jako pary. Na przykład w systemie implementującym język Scheme zgodnie ze standardem IEEE obiekt proceduralny może być zaimplementowany jako specjalny rodzaj „par”, która nie spełnia predykatu **pair?**. Na potrzeby symulacji **pointer-to-pair?** może być zaimplementowany jako **pair?**.

```

pair
  (assign oldcr (op vector-ref) (reg the-cars) (reg old))
  (test (op broken-heart?) (reg oldcr))
  (branch (label already-moved))
  (assign new (reg free)) ; nowe położenie pary
  ;; Uaktualnienie wskaznika free.
  (assign free (op +) (reg free) (const 1))
  ;; Skopiowanie car i cdr do nowej pamięci.
  (perform (op vector-set!)
    (reg new-cars) (reg new) (reg oldcr))
  (assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
  (perform (op vector-set!)
    (reg new-cdrs) (reg new) (reg oldcr))
  ;; Utworzenie złamanej serca.
  (perform (op vector-set!)
    (reg the-cars) (reg old) (const broken-heart))
  (perform
    (op vector-set!) (reg the-cdrs) (reg old) (reg new))
  (goto (reg relocate-continue))
already-moved
  (assign new (op vector-ref) (reg the-cdrs) (reg old))
  (goto (reg relocate-continue))

```

Na samym końcu procesu odśmiecania zamieniamy rolami starą i nową pamięć, zamieniając wskazniki: the-cars z new-cars i the-cdrs z new-cdrs. Będziemy wówczas gotowi do kolejnego odśmiecania, gdy znowu zabraknie pamięci.

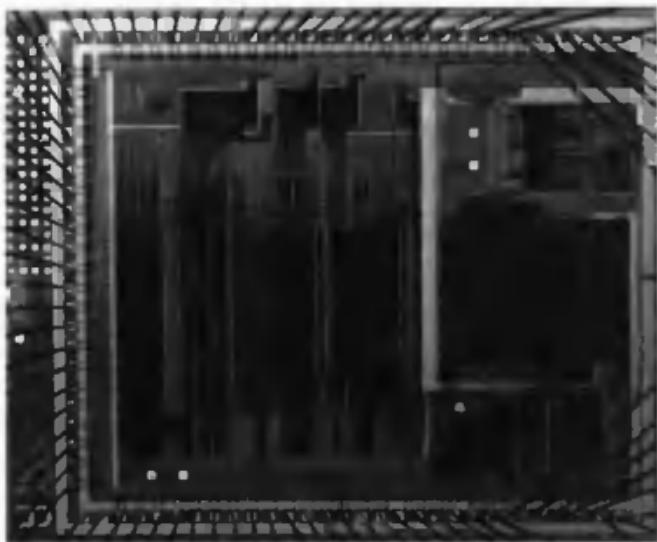
```

gc-flip
  (assign temp (reg the-cdrs))
  (assign the-cdrs (reg new-cdrs))
  (assign new-cdrs (reg temp))
  (assign temp (reg the-cars))
  (assign the-cars (reg new-cars))
  (assign new-cars (reg temp))

```

## 5.4. Evaluator z jawnie określonym sterowaniem

W podrozdziale 5.1 pokazaliśmy, jak można przekształcać proste programy w języku Scheme na opisy maszyn rejestrowych. Dokonamy teraz takiego przekształcenia na bardziej skomplikowanym programie — metacyklicznym evaluatorze przedstawionym w punktach 4.1.1–4.1.4 — który pokazuje, jak zachowanie interpretera języka Scheme może być opisane za pomocą procedur eval i apply. *Evaluator z jawnie określonym sterowaniem* (ang. *explicit-control evaluator*), który opracowujemy w niniejszym podrozdziale, obrazuje, w jaki sposób elementarne mechanizmy wywoływania procedur i przekazywa-



Rys. 5.16. Implementacja ewaluatora języka Scheme w postaci krzemowego układu scalonego

nia argumentów, używane w procesie obliczania, mogą być opisane za pomocą operacji na rejestrach i stosach. Ponadto ewaluator z jawnie określonym sterowaniem może służyć jako implementacja interpretera języka Scheme zapisana w języku podobnym do języka maszynowego tradycyjnych komputerów. Ewaluator ten może być wykonywany przez symulator maszyny rejestrowej, przedstawiony w podrozdziale 5.2. Może on też służyć za punkt wyjścia do budowy implementacji ewaluatora języka Scheme w języku maszynowym lub nawet do budowy wyspecjalizowanej maszyny obliczającej wyrażenia języka Scheme. Na rysunku 5.16 jest przedstawiona taka sprzętowa implementacja — krzemowy układ scalony, który działa jak ewaluator języka Scheme. Projektanci tego układu wyszli od określenia ścieżek danych i sterownika maszyny rejestrowej podobnej do ewaluatora opisanego w niniejszym podrozdziale i użyli do zaprojektowania układu scalonego programów automatycznego projektowania<sup>19</sup>.

### Rejestry i operacje

Projektując ewaluator z jawnie określonym sterowaniem, musimy sprecyzować operacje naszej maszyny rejestrowej. Opisaliśmy ewaluator metacykliczny za pomocą składni abstrakcyjnej, używając takich procedur, jak `quoted?` bądź

<sup>19</sup> Więcej informacji na temat tego układu scalonego i metod jego konstrukcji można znaleźć w [7].

`make-procedure`. Implementując maszynę rejestrową, moglibyśmy rozwijać te procedury jako ciągi elementarnych operacji pamięciowych na strukturach listowych i zaimplementować je w naszej maszynie rejestrowej. To jednak spowodowałoby, że evaluator byłby bardzo długi, a szczegóły zaciemniłyby jego podstawową strukturę. Dla przejrzystości przyjmujemy, że wśród pierwotnych operacji maszyny rejestrowej są procedury składniowe z punktu 4.1.2 oraz procedury implementujące środowiska i inne struktury danych potrzebne w trakcie wykonywania programu, przedstawione w punktach 4.1.3 i 4.1.4. Aby w pełni określić evaluator, który można by zaprogramować w niskopoziomowym języku maszynowym lub zaimplementować sprzętowo, zastąpimy te operacje prostszymi operacjami, używając implementacji struktur listowych opisanej w podrozdziale 5.3.

Nasza maszyna rejestrowa realizująca evaluator języka Scheme zawiera stos i siedem rejestrów: `exp`, `env`, `val`, `continue`, `proc`, `argl` i `unev`. Rejestr `exp` jest stosowany do przechowywania obliczanego wyrażenia, a `env` zawiera środowisko, w którym ma być wykonywane obliczenie. Po wykonaniu obliczenia `val` zawiera wynik obliczenia danego wyrażenia w danym środowisku. Rejestr `continue` jest wykorzystywany do zaimplementowania rekursji, tak jak to wyjaśniliśmy w punkcie 5.1.4. (Evaluator musi wywoływać rekurencyjnie sam siebie, gdyż obliczenie wyrażenia wymaga obliczenia jego podwyrażeń). Rejestry `proc`, `argl` i `unev` są używane przy obliczaniu kombinacji.

Nie podamy diagramu ścieżek danych pokazującego, w jaki sposób są połączone rejesty i operacje evaluatora, ani też nie podamy pełnej listy operacji maszynowych. Są one określone w niejawnym sposobie w sterowniku evaluatora, który przedstawimy szczegółowo.

#### 5.4.1. Trzon evaluatora z jawnie określonym sterowaniem

Głównym składnikiem evaluatora jest ciąg instrukcji zaczynający się od `eval-dispatch`. Odpowiada on procedurze `eval` evaluatora metacyklicznego opisanego w punkcie 4.1.1. Gdy sterownik zaczyna wykonywać podprogram `eval-dispatch`, oblicza on w środowisku określonym przez `env` wyrażenie określone przez `exp`. Gdy obliczenie się kończy, sterownik przechodzi do punktu wejścia zapamiętanego w `continue`, a rejestr `val` zawiera wartość wyrażenia. Tak jak w metacyklicznej procedurze `eval`, struktura podprogramu `eval-dispatch` odpowiada analizie przypadków różnych składniowych rodzajów obliczanych wyrażeń<sup>20</sup>.

<sup>20</sup> W naszym sterowniku rozdzielanie przypadków jest zapisane jako ciąg instrukcji `test` i `branch`. Alternatywnie mógłby on być zapisany w stylu programowania sterowanego danymi (i w prawdziwym systemie prawdopodobnie właśnie tak byłby zapisany) w celu uniknięcia wykonywania ciągłych porównań i ułatwienia definiowania nowych rodzajów wyrażeń. Maszyna zaprojektowana do wykonywania programów w Lispie prawdopodobnie zawierałaby instrukcję `dispatch-on-type`, która efektywnie wykonywałaby takie rozdzielanie sterowane danymi.

```

eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))

```

### Obliczanie wyrażeń prostych

Liczby i napisy (które są wyrażeniami „samoobliczającymi się”), zmienne, cytowania i lambda-wyrażenia nie mają podwyrażeń wymagających obliczania. W ich przypadku evaluator po prostu umieszcza właściwą wartość w rejestrze `val` i kontynuuje wykonywanie od punktu określonego przez `continue`. Obliczanie wyrażeń prostych jest realizowane przez następujący kod sterownika:

```

ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
            (reg unev) (reg exp) (reg env))
  (goto (reg continue))

```

Zwróćmy uwagę, w jaki sposób `ev-lambda` używa rejestrów `unev` i `exp` do przechowywania parametrów i treści lambda-wyrażenia, tak że mogą być one

przekazane do operacji `make-procedure` wraz ze środowiskiem przechowywanym w rejestrze `env`.

### Obliczanie zastosowań procedury

Zastosowanie procedury jest określone przez kombinację zawierającą operator i argumenty. Operator jest wyrażeniem, którego wartością jest procedura, a argumenty są podwyrażeniami, do wartości których należy zastosować daną procedurę. Metacykliczna procedura `eval` obsługuje zastosowania procedur, wywołując się rekurencyjnie w celu obliczenia każdego z elementów kombinacji, a następnie przekazując wyniki do procedury `apply`, która dokonuje faktycznego zastosowania procedury. Evaluator z jawnie określonym sterowaniem postępuje tak samo; te rekurencyjne wywołania są zaimplementowane za pomocą instrukcji `goto` oraz stosu służącego do zachowywania rejestrów, których zawartość jest odtwarzana po powrocie z rekurencyjnego wywołania. Przed każdym wywołaniem będziemy starannie rozpoznawać, które rejesty muszą być zachowane (ponieważ ich wartości będą potrzebne później)<sup>21</sup>.

Obliczanie zastosowania rozpoczynamy od obliczenia wartości operatora w celu uzyskania procedury, która ma być później zastosowana do obliczonych argumentów. Aby obliczyć wartość operatora, przenosimy go do rejestrów `exp` i przechodzimy do `eval-dispatch`. W rejestrze `env` jest przechowywane właściwe środowisko, w którym należy obliczyć wartość operatora. Zachowujemy jednak `env`, gdyż będzie nam później potrzebne do obliczania argumentów. Wydzielamy również argumenty do rejestrów `unev` i zachowujemy go na stosie. Rejestr `continue` ustawiamy tak, aby podprogram `eval-dispatch`, po obliczeniu operatora, kontynuował działanie od `ev-appl-did-operator`. Najpierw jednak zachowujemy starą wartość `continue`, która określa, gdzie sterownik powinien kontynuować działanie po zastosowaniu procedury.

```
ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

<sup>21</sup> Jest to ważny, acz subtelnny punkt w algorytmach tłumaczących programy z języka proceduralnego, takiego jak Lisp, na język maszyny rejestrowej. Alternatywą dla zachowywania tylko tego, co jest potrzebne, mogłoby być zapisywanie wszystkich rejestrów (z wyjątkiem `val`) przed każdym wywołaniem rekurencyjnym. Jest to nazywane dyscypliną *stosu ramek*. Takie podejście będzie działać, ale może wymagać zachowywania większej liczby rejestrów, niż to jest konieczne; może być to ważnym czynnikiem w systemie, w którym operacje na stosie są kosztowne. Zachowywanie rejestrów, których zawartość nie będzie później potrzebna, może powodować zatrzymywanie danych, które w przeciwnym razie zostałyby usunięte w ramach odśmiecania, zwalniając pamięć do ponownego wykorzystania.

Po obliczeniu wartości operatora przystępujemy do obliczania argumentów kombinacji i kumulowania ich w postaci listy przechowywanej w argl. Najpierw odtwarzamy nieobliczone jeszcze argumenty i środowisko. Rejestr argl inicjujemy jako pustą listę. Następnie przypisujemy do rejestru proc procedurę będącą wynikiem obliczenia operatora. Jeśli nie ma żadnych argumentów, to przechodzimy bezpośrednio do apply-dispatch. W przeciwnym razie zachowujemy wartość proc na stosie i rozpoczynamy pętlę obliczającą argumenty<sup>22</sup>:

```
ev-appl-did-operator
  (restore unev) ; argumenty
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val)) ; operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

W każdym kroku pętli jest obliczany jeden argument z listy unev, a wynik jest kumulowany w argl. Aby obliczyć wartość argumentu, umieszczamy go w rejestrze exp i przechodzimy do eval-dispatch, ustawiawszy uprzednio rejestr continue tak, aby wykonywanie było kontynuowane od fazy kumulacji argumentów. Najpierw jednak zachowujemy argumenty skumulowane do tej pory (przechowywane w argl), środowisko (przechowywane w env) oraz pozostałe do obliczenia argumenty (przechowywane w unev). Obliczanie ostatniego argumentu jest traktowane jako osobny przypadek, obsługiwany przez ev-appl-last-arg.

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
```

<sup>22</sup> Dołączamy do procedur operujących na strukturach danych evaluatora z punktu 4.1.3 następujące dwie procedury operujące na listach argumentów:

```
(define (empty-arglist) '())
(define (adjoin-arg arg arglist)
  (append arglist (list arg)))
```

Używamy również dodatkowej procedury składniowej sprawdzającej, czy argument jest ostatni na liście:

```
(define (last-operand? ops)
  (null? (cdr ops)))
```

```
(save unev)
(assign continue (label ev-appl-accumulate-arg))
(goto (label eval-dispatch))
```

Gdy argument zostanie obliczony, jego wartość jest kumulowana na liście przechowywanej w `arg1`. Następnie jest on usuwany z listy nieobliczonych argumentów `unev`, po czym odbywa się obliczanie kolejnych argumentów.

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore arg1)
  (assign arg1 (op adjoin-arg) (reg val) (reg arg1))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))
```

Ostatni argument jest obliczany inaczej. Nie ma potrzeby zachowywania środowiska bądź listy nieobliczonych jeszcze argumentów przed skokiem do `eval-dispatch`, gdyż nie będą one potrzebne po obliczeniu ostatniego argumentu. Dlatego też wracamy z obliczania jego wartości do specjalnego punktu wejścia `ev-appl-accum-last-arg`, gdzie jest odtwarzana lista argumentów, jest do niej dodawany nowy argument, odtwarzana jest zapisana procedura i jest ona stosowana do argumentów<sup>23</sup>.

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore arg1)
  (assign arg1 (op adjoin-arg) (reg val) (reg arg1))
  (restore proc)
  (goto (label apply-dispatch))
```

Szczegóły pętli obliczającej argumenty określają kolejność, w jakiej interpreter oblicza argumenty kombinacji (np. od lewej do prawej lub odwrotnie — zob. ćwiczenie 3.8). Kolejność ta nie jest określona przez evaluator metacykliczny, który przejmuje strukturę sterowania z działającego pod

---

<sup>23</sup> Optymalizacja polegająca na specjalnym traktowaniu ostatniego argumentu jest znana jako *rekursja ogonowa typu evis* (zob. [106]). Moglibyśmy jeszcze trochę bardziej efektywnie obliczać argumenty, gdybyśmy obliczanie pierwszego argumentu również traktowali jako odrębny przypadek. Pozwoliłoby to na odsunięcie inicjowania `arg1` do momentu po obliczeniu pierwszego argumentu, co pozwoliłoby uniknąć w tym przypadku zachowywania `arg1`. Komplilator przedstawiony w podrozdziale 5.5 wykonuje taką optymalizację. (Porównaj procedurę `construct-arglist` z punktu 5.5.3).

spodem języka Scheme, w którym jest zaimplementowany<sup>24</sup>. Ponieważ selektor `first-operand` (używany w `ev-appl-operand-loop` do wydzielania kolejnych argumentów z `unev`) jest zaimplementowany jako `car`, a selektor `rest-operands` jest zaimplementowany jako `cdr`, evaluator z jawnie określonym sterowaniem będzie obliczał argumenty kombinacji w kolejności od lewej do prawej.

### Zastosowanie procedury

Punkt wejścia `apply-dispatch` odpowiada procedurze `apply` evaluatora metacyklicznego. W momencie gdy dochodzimy do `apply-dispatch`, rejestr `proc` zawiera procedurę, którą należy zastosować, a `argl` zawiera listę obliczonych argumentów, do których należy ją zastosować. Zachowana wartość `continue` (początkowo przekazywana do `eval-dispatch` i zachowywana przez `ev-application`), która określa, gdzie należy wrócić z wynikiem zastosowania procedury, znajduje się na stosie. Po wykonaniu procedury sterowanie jest przekazywane do miejsca określonego przez zachowaną wartość `continue`, a aktualny wynik zastosowania procedury jest pamiętany w `val`. Tak jak w przypadku metacyklicznej procedury `apply`, należy rozważyć dwa przypadki. Procedura, którą należy zastosować, jest procedurą pierwotną albo złożoną.

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

Zakładamy, że każda operacja pierwotna jest tak zaimplementowana, że pobiera argumenty z `argl` i umieszcza wynik w `val`. Chcąc określić, w jaki sposób maszyna wykonuje operacje pierwotne, musielibyśmy podać ciąg instrukcji sterownika implementujący każdą z operacji pierwotnych i tak zorganizować `primitive-apply`, aby w zależności od zawartości `proc` wykonywane były instrukcje realizujące odpowiednią operację pierwotną. Ponieważ bardziej interesuje nas struktura procesu obliczania niż szczegółowo operacji pierwotnych, będziemy zamiast tego po prostu używać operacji `apply-primitive-procedure`, która stosuje procedurę określoną przez `proc` do argumentów z `argl`. Na potrzeby symulacji evaluatora za pomocą symulatora z podrozdziału 5.2 będziemy używać procedury `apply-primitive-procedure`, która w celu zastosowania operacji wywołuje dzia-

<sup>24</sup> Kolejność obliczania argumentów w evaluatorze metacyklicznym jest określona przez kolejność obliczania argumentów `cons` w procedurze `list-of-values` z punktu 4.1.1 (zob. ćwiczenie 4.1).

iąjący pod spodem system Scheme'a, tak jak to robiliśmy w evaluatorze metacyklicznym w punkcie 4.1.4. Po obliczeniu wyniku zastosowania operacji pierwotnej odtwarzamy rejestr `continue` i przechodzimy do wyznaczonego punktu wejścia.

```
primitive-apply
  (assign val (op apply-primitive-procedure)
            (reg proc)
            (reg arg1))
  (restore continue)
  (goto (reg continue))
```

Chcąc zastosować procedurę złożoną, postępujemy tak jak w evaluatorze metacyklicznym. Tworzymy ramkę, która wiąże parametry procedury z jej argumentami, używamy tej ramki do rozszerzenia środowiska określonego przez procedurę i obliczamy w tym środowisku ciąg wyrażeń stanowiący treść procedury. Podprogram `ev-sequence`, opisany w punkcie 5.4.2, oblicza ciąg wyrażeń.

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
            (reg unev) (reg arg1) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

`Compound-apply` to jedyne miejsce w interpreterze, gdzie do rejestrów `env` jest przypisywana nowa wartość. Tak jak w evaluatorze metacyklicznym, nowe środowisko jest tworzone ze środowiska określonego przez procedurę oraz listy argumentów i listy odpowiadających im zmiennych, które należy związać.

#### 5.4.2. Obliczanie ciągów wyrażeń i rekursja ogonowa

Fragment evaluatora z jawnie określonym sterowaniem zaczynający się od etykiety `ev-sequence` stanowi odpowiednik procedury `eval-sequence` evaluatora metacyklicznego. Przetwarza on ciągi wyrażeń pochodzące z treści procedur lub jawnie określonych wyrażeń `begin`.

Jawnie określone wyrażenia `begin` są obliczane w ten sposób, że ciąg wyrażeń, które należy obliczyć, jest umieszczany w `unev`, zawartość rejestrów `continue` jest zachowywana na stosie i wykonywany jest skok do `ev-sequence`.

```
ev-begin
  (assign unev (op begin-actions) (reg exp))
```

```
(save continue)
(goto (label ev-sequence))
```

Niejawne ciągi wyrażeń pochodzące z treści procedur są przetwarzane w ten sposób, że w compound-apply wykonywany jest skok do ev-sequence (w tym momencie wartość continue znajduje się już na stosie, gdyż została tam umieszczona w ev-application).

Punkty wejścia ev-sequence i ev-sequence-continue tworzą pętlę, która oblicza kolejno każde wyrażenie w ciągu. Lista nieobliczonych jeszcze wyrażeń jest pamiętana w unev. Przed obliczeniem każdego wyrażenia sprawdzamy, czy w ciągu są jeszcze jakieś dodatkowe wyrażenia do obliczenia. Jeśli tak, to zachowujemy resztę nieobliczonych jeszcze wyrażeń (pamiętaną w unev) oraz środowisko, w którym należy je obliczać (pamiętane w env), i wywołujemy eval-dispatch w celu obliczenia wyrażenia. Zawartość dwóch zachowanych rejestrów jest odtwarzana po powrocie z obliczania wyrażenia, w miejscu oznaczonym etykietą ev-sequence-continue.

Ostatnie wyrażenie w ciągu jest traktowane inaczej, w punkcie wejścia ev-sequence-last-exp. Ponieważ nie ma już po nim więcej wyrażeń do obliczania, nie musimy przed wywołaniem eval-dispatch zachowywać rejestrów unev i env. Wartością całego ciągu jest wartość ostatniego wyrażenia, więc po jego obliczeniu nie musimy nic robić z wyjątkiem kontynuowania obliczeń od punktu wejścia przechowywanego w danej chwili na stosie (zachowanego przez ev-application lub ev-begin). Zamiast ustawać continue w eval-dispatch tak, aby powracać tam, a następnie odtwarzając continue ze stosu i kontynuować obliczenia od tego punktu wejścia, odtwarzamy continue ze stosu przed skokiem do eval-dispatch, dzięki czemu po obliczeniu wyrażenia eval-dispatch kontynuuje obliczenia od tego punktu wejścia.

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

## Rekursja ogonowa

W rozdziale 1 powiedzieliśmy, że proces opisany przez taką procedurę jak

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

jest procesem iteracyjnym. Mimo że procedura jest składniowo rekurencyjna (zdefiniowana za pomocą siebie samej), logicznie rzecz biorąc, nie jest konieczne, aby evaluator zachowywał informacje, przechodząc od jednego wywołania `sqrt-iter` do następnego<sup>25</sup>. Evaluator, który potrafi wykonywać takie procedury jak `sqrt-iter` bez konieczności zwiększenia ilości zachowanych informacji, w miarę jak procedura wywołuje ciągle samą siebie, jest nazywany evaluatorem z *rekursją ogonową*. Metacykliczna implementacja evaluatora z rozdziału 4 nie określa, czy evaluator jest z rekursją ogonową, ponieważ evaluator przejmuje mechanizm zachowywania stanów z działającego pod spodem języka Scheme. Jednak w przypadku evaluatora z jawnie określonym sterowaniem możemy prześledzić proces obliczania i zobaczyć, kiedy wywołanie procedury powoduje ostatecznie gromadzenie się informacji na stosie.

Nasz evaluator jest evaluatorem z rekursją ogonową, ponieważ w celu obliczenia ostatniego wyrażenia w ciągu skaczemy bezpośrednio do `eval-dispatch` bez zachowywania żadnych informacji na stosie. W związku z tym, obliczając ostatnie wyrażenie w ciągu — nawet jeśli jest to wywołanie procedury (tak jak w `sqrt-iter`, gdzie ostatnim wyrażeniem w treści procedury jest wyrażenie `if`, które sprowadza się do wywołania `sqrt-iter`) — nie powodujemy gromadzenia się żadnych informacji na stosie<sup>26</sup>.

Gdybyśmy nie pomyśleli o wykorzystaniu faktu, że w tym przypadku nie trzeba zachowywać informacji na stosie, moglibyśmy zaimplementować `eval-sequence` tak, że wszystkie wyrażenia w ciągu traktowalibyśmy w ten sam sposób — zachowując rejesty, obliczając wyrażenia, powracając z wywołania, odtwarzając rejesty i powtarzając to, dopóki wszystkie wyrażenia nie będą obliczone<sup>27</sup>:

<sup>25</sup> W podrozdziale 5.1 widzieliśmy, jak można zaimplementować taki proces za pomocą maszyny rejestrowej bez stosu; stan procesu był pamiętany w ustalonym zestawie rejestrów.

<sup>26</sup> Taka implementacja rekursji ogonowej w `ev-sequence` jest jedną z wielu dobrze znanych technik optymalizacji używanych w wielu kompilatorach. Kompilując procedurę, która jest zakończona wywołaniem procedury, można zastąpić to wywołanie skokiem do punktu wejścia tej procedury. Wbudowując taką strategię w interpreter, jak to zrobiliśmy w niniejszym punkcie, wprowadzamy tę optymalizację w jednolity sposób do całego języka.

<sup>27</sup> `No-more-exp?` możemy zdefiniować w następujący sposób:

```
(define (no-more-exp? seq) (null? seq))
```

```

ev-sequence
  (test (op no-more-exps?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))

ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))

ev-sequence-end
  (restore continue)
  (goto (reg continue))

```

Może to wyglądać na pomniejszą zmianę w poprzednio przedstawionym kodzie obliczającym ciąg wyrażeń — jedyna różnica polega na tym, że pozostajemy w cyklu zachowaj-odtwórz aż do ostatniego wyrażenia w ciągu. Interpreter będzie nadal dawał takie same wyniki dla dowolnych wyrażeń. Jednak taka zmiana jest fatalna w skutkach dla rekursji ogonowej, ponieważ musimy teraz po obliczeniu ostatniego wyrażenia w ciągu powrócić z wywołania w celu odtworzenia (niepotrzebnych) wartości rejestrów. Te dodatkowo zachowywane wartości będą się gromadzić w trakcie zagnieźdzania wywołań procedury. W rezultacie procesy takie jak `sqrt-iter` będą wymagać pamięci proporcjonalnej do liczby iteracji zamiast stałej pamięci. Taka różnica może być istotna. Dysponując rekursją ogonową, możemy na przykład wyrazić pętlę nieskończoną jedynie za pomocą mechanizmu wywoływania procedur:

```

(define (count n)
  (newline)
  (display n)
  (count (+ n 1)))

```

Bez rekursji ogonowej dla takiej procedury zabrakłoby w końcu pamięci na stosie i wyrażanie prawdziwie nieskończonych pętli wymagałoby dodatkowych mechanizmów sterowania, innych niż wywoływanie procedur.

### 5.4.3. Instrukcje warunkowe, przypisania i definicje

Tak jak w evaluatorze metacyklicznym, obsługiwanie form specjalnych polega na selektywnym obliczaniu fragmentów wyrażenia. W przypadku wyrażenia `if` musimy obliczyć predykat i w zależności od jego wartości musimy zdecydować, czy obliczać następnik, czy też alternatywę.

Przed obliczeniem predykatu zachowujemy samo wyrażenie `if`, aby móc później wydzielić z niego następnik lub alternatywę. Zachowujemy również środowisko, które będzie nam później potrzebne w trakcie obliczania następnika lub alternatywy, a także zachowujemy zawartość rejestru `continue`, która będzie nam później potrzebna do powrotu do obliczania wyrażenia oczekującego na wartość wyrażenia `if`.

```
ev-if
  (save exp) ; zachowaj wyrażenie na później
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch)) ; oblicz predykat
```

Gdy następuje powrót z obliczenia predykatu, sprawdzamy, czy był on spełniony, czy nie, i w zależności od tego umieszczamy w `exp` następnik lub alternatywę i skaczemy do `eval-dispatch`. Zwróćmy uwagę, że odtwarzając `env` i `continue`, ustawiamy dla `eval-dispatch` właściwe środowisko i adres powrotu do właściwego miejsca, do którego powinna trafić wartość wyrażenia `if`.

```
ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))
```

### Przypisania i definicje

Przypisania są obsługiwane przez `ev-assignment`, gdzie skaczemy z `eval-dispatch`, mając zapamiętane wyrażenie przypisania w `exp`. Podprogram `ev-assignment` najpierw oblicza część wyrażenia określającą wartość, a następnie umieszcza uzyskaną wartość w środowisku. Zakładamy, że `set-variable-value!` jest dostępna jako operacja maszynowa.

```
ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev) ; zachowaj zmienną na później
  (assign exp (op assignment-value) (reg exp))
  (save env)
```

```

(save continue)
(assign continue (label ev-assignment-1))
(goto (label eval-dispatch)) ; oblicz wartość przypisania
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
    (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue)))

```

Definicje są obsługiwane w podobny sposób:

```

ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev) ; zachowaj zmienną na później
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch)) ; oblicz definiowaną wartość
ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
    (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue)))

```

### Ćwiczenie 5.23

Rozszerz tak ewaluator, aby przetwarzał wyrażenia pochodne, takie jak cond, let itp. (punkt 4.1.2). Możesz „oszukać” i założyć, że przekształcenia składniowe, takie jak cond->if, są dostępne w formie operacji maszynowych<sup>28</sup>.

### Ćwiczenie 5.24

Zaimplementuj cond jako nową podstawową formę specjalną, bez sprowadzania jej do if. Będziesz musiał utworzyć pętlę sprawdzającą predykaty kolejnych klauzul cond aż do momentu znalezienia prawdziwej, a następnie użyć ev-sequence do obliczenia następnika klauzuli.

<sup>28</sup> Tak naprawdę nie jest to oszustwo. W prawdziwej implementacji zbudowanej od zera, w fazie składniowej, przed rozpoczęciem wykonywania, użylibyśmy naszego ewaluatora z jawnie określonym sterowaniem do zinterpretowania programu w języku Scheme, który wykonuje przekształcenia na poziomie źródłowym, takie jak cond->if.

### Ćwiczenie 5.25

Opierając się na konstrukcji leniwego evaluatora z podrozdziału 4.2, tak zmodyfikuj evaluator, aby stosował normalną kolejność obliczania.

#### 5.4.4. Uruchamianie evaluatora

Wraz z implementacją evaluatora z jawnie określonym sterowaniem dochodzimy do końca procesu rozpoczętego w rozdziale 1, w trakcie którego badaliśmy kolejno coraz bardziej precyzyjne modele obliczeń. Zaczeliśmy od stosunkowo nieformalnego modelu podstawieniowego, rozszerzyliśmy go w rozdziale 3, uzyskując model środowiskowy, który pozwolił nam na operowanie pojęciami stanu i zmiany. W metacyklicznym evaluatorze z rozdziału 4 użyliśmy samego języka Scheme jako języka do bardziej wyraźnego opisu struktury środowisk tworzonej w trakcie obliczeń. Teraz, zapoznając się z maszynami rejestrówymi, przyjrzeliśmy się bliżej mechanizmom zarządzania pamięcią, przekazywania argumentów i sterowania w evaluatorze. Na każdym z kolejnych poziomów opisu musieliśmy poruszać problemy i rozstrzygać niejasności, które nie były widoczne na poprzednim, mniej precyzyjnym poziomie opisu obliczeń. Aby zrozumieć działanie evaluatora z jawnie określonym sterowaniem, możemy go zasymulować i monitorować jego wydajność.

W naszej maszynie evaluatora zainstalujemy pętlę sterującą. Odgrywa ona rolę procedury `driver-loop` z punktu 4.1.4. Evaluator będzie wielokrotnie wypisywał znak zachęty, wczytywał wyrażenie, obliczał je, skacząc do `eval-dispatch`, i wypisywał wynik. Następujące instrukcje stanowią początek ciągu instrukcji sterownika evaluatora z jawnie określonym sterowaniem<sup>29</sup>:

```
read-eval-print-loop
  (perform (op initialize-stack))
  (perform
    (op prompt-for-input) (const ";;; EC-Eval wprowadź wyrażenie:"))
    (assign exp (op read))
    (assign env (op get-global-environment))
    (assign continue (label print-result))
  (goto (label eval-dispatch))
```

<sup>29</sup> Zakładamy tutaj, że `read` oraz różne operacje wypisujące są dostępne jako pierwotne operacje maszynowe, co jest bardzo użyteczne na potrzeby naszej symulacji, ale w praktyce całkowicie nirealistyczne. Są to w rzeczywistości bardzo złożone operacje. W praktyce byłyby one zaimplementowane za pomocą niskopoziomowych operacji wejścia-wyjścia, takich jak przesyłanie pojedynczych znaków do i z urządzeń.

Aby zrealizować operację `get-global-environment`, wprowadzamy definicje

```
(define the-global-environment (setup-environment))
(define (get-global-environment)
  the-global-environment)
```

```

print-result
  (perform
    (op announce-output) (const ";; EC-Eval wartość:"))
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))

```

Gdy napotykamy w procedurze błąd (taki jak „nieznany rodzaj procedury” sygnalizowany przez `apply-dispatch`), wypisujemy komunikat błędu i powracamy do pętli sterującej<sup>30</sup>.

```

unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))

```

```

unknown-procedure-type
  (restore continue) ; uprągtnij stos (po apply-dispatch)
  (assign val (const unknown-procedure-type-error))
  (goto (label signal-error))

```

```

signal-error
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))

```

Na potrzeby symulacji stos inicjujemy w każdym kroku pętli sterującej, gdyż może on nie być pusty po tym, jak obliczenie zostaje przerwane przez błąd (taki jak np. nieznana zmienna)<sup>31</sup>.

Jeśli połączymy wszystkie fragmenty kodu przedstawione w punktach 5.4.1–5.4.4, to możemy stworzyć model maszyny ewaluatora, który możemy uruchomić za pomocą symulatora maszyn rejestrowych z podrozdziału 5.2.

```

(define eceval
  (make-machine
    '(exp env val proc argl continue unev)
    eceval-operations
    '(
      read-eval-print-loop
      cały sterownik maszyny, taki jak podany powyżej
    )))

```

Musimy zdefiniować w języku Scheme procedury symulujące operacje używane w ewaluatorze jako pierwotne. Są to te same procedury, których używaliśmy

<sup>30</sup> Istnieją inne błędy, które chcielibyśmy, żeby interpreter obsługiwał, ale nie jest to takie proste. Zobacz ćwiczenie 5.30.

<sup>31</sup> Moglibyśmy inicjować stos tylko po wystąpieniu błędu, ale inicjowanie stosu w pętli sterującej okaże się dogodne ze względu na monitorowanie wydajności ewaluatora, jak to za chwilę opiszymy.

w evaluatorze metacyklicznym z podrozdziału 4.1, wraz z kilkoma dodatkowymi procedurami zdefiniowanymi w przypisach w podrozdziale 5.4.

```
(define eceval-operations
  (list (list 'self-evaluating? self-evaluating)
        'pełna lista operacji maszyny eceval)))
```

Na koniec możemy zainicjować środowisko globalne i uruchomić evaluator:

```
(define the-global-environment (setup-environment))
```

```
(start eceval)
```

```
;; EC-Eval wprowadź wyrażenie:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;; EC-Eval wartość:
ok

;; EC-Eval wprowadź wyrażenie:
(append '(a b c) '(d e f))
;; EC-Eval wartość:
(a b c d e f)
```

Oczywiście obliczanie wyrażeń w ten sposób zajmuje dużo więcej czasu niż bezpośrednie wprowadzenie ich do interpretera języka Scheme, ze względu na wiele poziomów symulacji. Nasze wyrażenia są obliczane przez maszynę evaluatora z jawnie określonym sterowaniem, która jest symulowana przez program w języku Scheme obliczany przez interpreter tego języka.

### Monitorowanie wydajności evaluatora

Symulacja może być potężnym narzędziem wspomagającym implementowanie evaluatorów. Symulacje nie tylko ułatwiają badanie różnych wariantów projektów maszyn rejestrowych, lecz także ułatwiają monitorowanie wydajności symulowanego evaluatora. Istotnym czynnikiem wydajnościowym jest na przykład to, jak efektywnie evaluator używa stosu. Możemy badać liczbę operacji stosowych potrzebnych do obliczenia rozmaitych wyrażeń, używając do uruchamiania evaluatora maszyn rejestrowych wersji symulatora, która gromadzi dane statystyczne dotyczące wykorzystania stosu, oraz dodając w punkcie wejścia evaluatora `print-result` instrukcję wypisującą zgromadzone statystyki:

```
print-result
  (perform (op print-stack-statistics)) ; dodana instrukcja
```

```
(perform
  (op announce-output) (const ";;; EC-Eval wartość:")
  ... ; tak samo jak poprzednio
```

Interakcje z ewaluatorem będą teraz wyglądać tak jak ta:

```
;;; EC-Eval wprowadź wyrażenie:
(define (factorial n)
  (if (= n 1)
    1
    (* (factorial (- n 1)) n)))
(łączna liczba włożeń = 3 maksymalny rozmiar = 3)
;;; EC-Eval wartość:
ok

;;; EC-Eval wprowadź wyrażenie:
(factorial 5)
(łączna liczba włożeń = 144 maksymalny rozmiar = 28)
;;; EC-Eval wartość:
120
```

Zwróćmy uwagę, że pętla sterująca ewaluatora inicjuje stos na początku każdej interakcji, tak więc wypisywane statystyki odnoszą się wyłącznie do operacji stosowych wykonanych w trakcie obliczania ostatnio wprowadzonego wyrażenia.

### Ćwiczenie 5.26

Użyj monitorowanego stosu do zbadania własności rekursji ogonowej ewaluatora (punkt 5.4.2). Uruchom ewaluator i zdefiniuj iteracyjną procedurę `factorial` z punktu 1.2.1:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Uruchom tę procedurę dla kilku niewielkich wartości  $n$ . Dla każdej z tych wartości zanotuj, jaki był maksymalny rozmiar stosu i jaką liczbę operacji wkładania na stos wykonano w trakcie obliczania  $n!$ .

- (a) Okaże się, że maksymalny rozmiar stosu w trakcie obliczania  $n!$  nie zależy od  $n$ . Jaki jest ten rozmiar?
- (b) Na podstawie zgromadzonych danych podaj wzór określający w zależności od  $n$ , dla dowolnego  $n \geq 1$ , łączną liczbę operacji wkładania na stos wykonywanych w trakcie obliczania  $n!$ . Zauważ, że liczba wykonywanych operacji jest liniową funkcją  $n$ , a więc można ją wyznaczyć na podstawie wartości w dwóch punktach.

**Ćwiczenie 5.27**

Dla porównania z ćwiczeniem 5.26, zbadaj zachowanie następującej procedury obliczającej silnię rekurencyjnie:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

Uruchamiając tę procedurę z monitorowaniem stosu, określ w zależności od  $n$ , dla  $n \geq 1$ , maksymalny rozmiar stosu oraz łączną liczbę operacji wkładania na stos wykonywanych w trakcie obliczania  $n!$ . (Ponownie będą to funkcje liniowe). Podsumuj wyniki przeprowadzonych eksperymentów, wypełniając następującą tabelkę odpowiednimi wyrażeniami zależnymi od  $n$ :

	Maksymalny rozmiar	Liczba operacji wkładania
Rekurencyjna silnia		
Iteracyjna silnia		

Maksymalny rozmiar stosu jest miarą ilości pamięci potrzebnej evaluatorowi do wykonania obliczeń, a liczba operacji wkładania jest ściśle związana z potrzebnym czasem.

**Ćwiczenie 5.28**

Zmodyfikuj definicję evaluatora, zmieniając eval-sequence tak, jak opisaliśmy to w punkcie 5.4.2, aby evaluator nie używał rekursji ogonowej. Wykonaj ponownie eksperymenty z ćwiczeń 5.26 i 5.27 i pokaż, że obie wersje procedury factorial wymagają teraz pamięci rosnącej liniowo ze względu na wartość argumentu.

**Ćwiczenie 5.29**

Monitoruj operacje stosowe w rozgałęziającym się rekurencyjnie procesie obliczania liczb Fibonacciego:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

(a) Podaj wzór określający w zależności od  $n$ , dla  $n \geq 2$ , maksymalny rozmiar stosu w trakcie obliczania  $\text{Fib}(n)$ . Wskazówka: W punkcie 1.2.2 stwierdziliśmy, że pamięć potrzebna dla tego procesu rośnie liniowo ze względu na  $n$ .

(b) Podaj wzór określający łączną liczbę operacji wkładania na stos wykonywanych w trakcie obliczania  $\text{Fib}(n)$ , dla  $n \geq 2$ . Powinno się okazać, że liczba tych operacji (która jest ściśle związana z czasem potrzebnym do wykonania obliczenia) rośnie wykładniczo ze względu na  $n$ . Wskazówka: Niech  $S(n)$  będzie liczbą operacji wkładania na stos wykonywanych w trakcie obliczania  $\text{Fib}(n)$ . Powinieneś móc stwierdzić, że  $S(n)$  można wyrazić wzorem zależnym od  $S(n-1)$ ,  $S(n-2)$  i pewnej ustalonej stałej

„narzutu”  $k$  niezależnej od  $n$ . Podaj ten wzór i stałą  $k$ . Następnie pokaż, że  $S(n)$  można wyrazić jako  $a \text{Fib}(n+1) + b$ , i podaj wartości współczynników  $a$  i  $b$ .

### Ćwiczenie 5.30

Nasz evaluator aktualnie wykrywa i zgłasza tylko dwa rodzaje błędów: nieznany rodzaj wyrażeń i nieznany rodzaj procedury. Inne błędy spowodują przerwanie pętli wczytaj-oblicz-wypisz evaluatora. Gdy uruchamiamy evaluator za pomocą symulatora maszyn rejestrowych, błędy te są wykrywane przez działający pod spodem system języka Scheme. Jest to odpowiednik awarii komputera na skutek błędu spowodowanego przez program użytkownika<sup>32</sup>. Opracowanie prawdziwego systemu wykrywania błędów to poważne zadanie, jednak zrozumienie, co się tam dzieje, jest stanowczo warte tego wysiłku.

- (a) Błędy pojawiające się w trakcie obliczeń, takie jak próba sięgnięcia do zmiennej wolnej, mogą być wychwytywane poprzez taką zmianę operacji wyszukiwania wartości w środowisku, że jej wynikiem może być wyróżniony kod błędu, który nie może być wartością żadnej zmiennej użytkownika. Evaluator może sprawdzić, czy wynik tej operacji jest kodem błędu, i wykonać odpowiedni skok do `signal-error`. Znajdź wszystkie miejsca w evaluatorze, gdzie konieczne są takie zmiany, i wprowadź je. Jest to mnóstwo roboty.
- (b) Dużo gorszy jest problem obsługi błędów zgłaszanych na skutek zastosowania procedur pierwotnych, takich jak próba dzielenia przez zero lub próba wydzielenia `car` symbolu. W wysokiej jakości profesjonalnie napisanym systemie, w ramach każdego zastosowania operacji pierwotnej sprawdzane są odpowiednie warunki bezpieczeństwa. Na przykład każde wywołanie `car` mogłoby najpierw sprawdzić, czy argument jest parą. Jeśli nie jest on parą, to wynikiem zastosowania operacji byłby wyróżniony kod błędu, który zostałby przekazany do evaluatora, a ten zgłosiłby błąd. Moglibyśmy tak zorganizować nasz symulator maszyn rejestrowych, aby każda procedura pierwotna sprawdzała, czy można ją zastosować, i przekazywała odpowiedni wyróżniony kod błędu w przypadku niepowodzenia. Następnie podprogram `primitive-apply` w evaluatorze może sprawdzać, czy pojawił się kod błędu, i w razie potrzeby skakać do `signal-error`. Zrealizuj taką strukturę i uruchom ją. Jest to poważne zadanie.

## 5.5. Kompilacja

Evaluator z jawnie określonym sterowaniem z podrozdziału 5.4 to maszyna rejestrowa, której sterownik interpretuje programy w języku Scheme. W najbliższym podrozdziale zobaczymy, jak można uruchamiać programy w języku Scheme na maszynie rejestrowej, której sterownik nie jest interpreterem tego języka.

<sup>32</sup> Godny ubolewania jest fakt, że jest to normalny stan rzeczy w tradycyjnych, opartych na kompilatorach systemach języków takich jak C. W systemie UNIX<sup>TM</sup> następuje zrzut obrazu pamięci, a system DOS/Windows<sup>TM</sup> popada w katatonię. System Macintosh<sup>TM</sup> wyświetla rysunek eksplodującej bomby i oferuje — jeśli mamy szczęście — możliwość ponownego uruchomienia komputera.

Maszyna ewaluatora z jawnie określonym sterowaniem jest uniwersalna — może wykonywać dowolne procesy obliczeniowe, które można opisać w języku Scheme. Sterownik ewaluatora dyryguje użyciem swoich ścieżek danych w celu wykonania zadanych obliczeń. Tak więc ścieżki danych ewaluatora są uniwersalne — wystarczają do wykonania dowolnego obliczenia, jakie sobie zażyczymy, jeśli dostarczymy odpowiedni sterownik<sup>33</sup>.

Dostępne w handlu komputery ogólnego przeznaczenia to maszyny rejestrów, których serce stanowi zestaw rejestrów i operacji, tworzący efektywny i dogodny w użyciu uniwersalny zestaw ścieżek danych. Sterownik maszyny ogólnego przeznaczenia to interpreter języka maszyny rejestrowej, takiego jak ten, którego używaliśmy. Język ten jest nazywany *językiem rodzimym* (ang. *native language*) maszyny lub po prostu *językiem maszynowym* (ang. *machine language*). Programy napisane w języku maszynowym to ciągi instrukcji operujących na ścieżkach danych maszyny. Na przykład ciąg instrukcji ewaluatora z jawnie określonym sterowaniem możemy traktować jak program w języku maszynowym komputera ogólnego przeznaczenia, a nie jak opis sterownika maszyny będącej wyspecjalizowanym interpreterem.

Istnieją dwie powszechnie stosowane metody przechodzenia od języków wysokiego poziomu do języków maszyn rejestrowych. Ewaluator z jawnie określonym sterowaniem ilustruje metodę polegającą na interpretacji. Interpreter napisany w języku rodzimym maszyny tak steruje maszyną, aby wykonywała ona programy napisane w języku (nazywanym *językiem źródłowym*), który może się różnić od języka rodzimego maszyny wykonującej obliczenia. Procedury pierwotne języka źródłowego są zaimplementowane w postaci biblioteki podprogramów napisanych w rodzimym języku danej maszyny. Interpretowany program (nazywany *programem źródłowym*) jest reprezentowany jako struktura danych. Interpreter przegląda tę strukturę danych, analizując program źródłowy. Równocześnie symuluje pożądane zachowanie programu źródłowego, wyowołując odpowiednie podprogramy pierwotne z biblioteki.

W niniejszym podrozdziale badamy alternatywną metodę polegającą na *kompilacji*. Kompilator, dla danego języka źródłowego i maszyny, tłumaczy program źródłowy na równoważny program (nazywany *programem wynikowym*) zapisany w języku rodzimym maszyny. Kompilator, który implementujemy w tym podrozdziale, tłumaczy programy napisane w języku Scheme na ciągi instrukcji przeznaczone do wykonania przy wykorzystaniu ścieżek danych ewaluatora z jawnie określonym sterowaniem<sup>34</sup>.

<sup>33</sup> Jest to stwierdzenie teoretyczne. Nie twierdzimy, że ścieżki danych ewaluatora są szczególnie dogodne lub wydajne w przypadku komputera ogólnego przeznaczenia. Nie są one na przykład zbyt dobre do implementowania bardzo wydajnych obliczeń zmiennopozycyjnych ani do obliczeń intensywnie operujących na wektorach bitów.

<sup>34</sup> W rzeczywistości maszyna wykonująca skompilowany program może być prostsza niż maszyna interpretera, ponieważ nie będziemy używać rejestrów `exp` i `unev`. Interpreter używał ich

W porównaniu z interpretacją komplikacja może przynieść duże zwiększenie efektywności wykonywania programów, co wyjaśnimy w poniższym przeglądzie kompilatora. Z drugiej strony, interpreter dostarcza potężniejszego środowiska do rozwijania i odpluskwięcia programów, ponieważ program źródłowy jest dostępny w trakcie wykonywania, można więc go badać i modyfikować. Ponadto, ponieważ cała biblioteka operacji pierwotnych jest dostępna, nowe programy mogą być konstruowane i dodawane do systemu w trakcie odpluskwięcia.

Zważywszy na wzajemnie się uzupełniające zalety komplikacji i interpretacji, współczesne środowiska programistyczne stosują mieszaną strategię. Ogólnie mówiąc, interpreterzy Lispu są tak zbudowane, że procedury interpretowane i skompilowane mogą się nawzajem wywoływać. Pozwala to programiście na skompilowanie tych części programu, które uważa za bezbłędne, umożliwiając skorzystanie z efektywności komplikacji przy jednoczesnym zachowaniu interpretowanego trybu wykonywania tych części programu, które nieustannie się zmieniają na skutek interakcyjnego ich rozwijania i odpluskwięcia. W punkcie 5.5.7, po zaimplementowaniu kompilatora, pokażemy, jak go sprząć z naszym interpreterem, aby utworzył zintegrowany interpretującą-kompilującą system programistyczny.

## Przegląd kompilatora

Nasz kompilator w znacznym stopniu przypomina nasz interpreter zarówno pod względem struktury, jak i wykonywanych funkcji. W związku z tym mechanizmy używane przez kompilator do analizy wyrażeń będą podobne do używanych przez interpreter. Ponadto, aby ułatwić scalenie kodu skompilowanego i interpretowanego, tak zaprojektujemy kompilator, aby generował kod używający rejestrów zgodnie z tymi samymi konwencjami co interpreter — środowisko będzie przechowywane w rejestrze `env`, lista argumentów będzie gromadzona w `arg1`, procedury do zastosowania będą pamiętane w `proc`, wyniki procedur będą przekazywane w `val`, a adresy powrotu będą przechowywane w `continue`. Ogólnie rzecz biorąc, kompilator tłumaczy program źródłowy na program wynikowy, który wykonuje zasadniczo te same operacje na rejestrach, które wykonywałby interpreter obliczający dany program źródłowy.

Opis ten wskazuje sposób implementacji prostego kompilatora: Przeglądamy wyrażenie w ten sam sposób co interpreter. Gdy napotykamy instrukcję

---

do przechowywania fragmentów nieobliczonych jeszcze wyrażeń. Jednak w przypadku kompilatora wyrażenia te zostaną wbudowane w skompilowany kod wykonywany przez maszynę rejestrową. Z tego samego powodu nie potrzebujemy operacji dotyczących składni wyrażeń. Jednak skompilowany kod będzie używał kilku dodatkowych operacji maszynowych (służących do reprezentowania obiektów proceduralnych), które nie występowały w maszynie ewaluatora z jawnie określonym sterowaniem.

rejestrowe, które interpreter wykonałby w trakcie obliczania wyrażenia, nie wykonujemy ich, lecz gromadzimy w formie ciągu. Tak powstały ciąg instrukcji będzie kodem wynikowym. Zwróćmy uwagę na przewagę, jaką komplikacja ma nad interpretacją pod względem efektywności. Za każdym razem gdy interpreter oblicza wyrażenie — na przykład (f 84 96) — klasyfikuje on je (odkrywając, że jest to zastosowanie procedury) oraz szuka końca listy argumentów (odkrywając, że są dwa argumenty). W przypadku kompilatora wyrażenie jest analizowane tylko raz, w momencie generowania ciągu instrukcji, w trakcie komplikacji. Kod wynikowy generowany przez kompilator zawiera tylko instrukcje obliczające operator i dwa argumenty, konstruujące listę argumentów i stosujące procedurę (z proc) do argumentów (z arg1).

Jest to ten sam rodzaj optymalizacji, który zaimplementowaliśmy w ewaluatorze z fazą analizy w punkcie 4.1.7. Jednak w skompilowanym kodzie pojawiają się dalsze możliwości zwiększenia efektywności. Interpreter wykonuje proces, który musi dać się zastosować do dowolnego wyrażenia z języka. W odróżnieniu od tego dany segment skompilowanego kodu ma wykonywać pewne określone wyrażenie. Może to sprawiać dużą różnicę na przykład w użyciu stosu do zachowywania rejestrów. Gdy interpreter oblicza wyrażenie, musi być przygotowany na wszelkie ewentualności. Przed obliczeniem podwyrażenia interpreter zachowuje zawartość wszystkich rejestrów, które będą potrzebne później, ponieważ podwyrażenie może wymagać dowolnych obliczeń. Z kolei kompilator może wykorzystać strukturę konkretnego, przetwarzanego wyrażenia, aby wygenerować kod, który nie wykonuje niepotrzebnych operacji na stosie.

Rozważmy na przykład kombinację (f 84 96). Zanim interpreter obliczy operator kombinacji, przygotowuje się do tego obliczenia, zachowując zawartości rejestrów zawierających argumenty i środowisko, które będą potrzebne później. Następnie interpreter oblicza operator, umieszczając wynik w val, odtwarza zachowane rejesty i na koniec przenosi wynik z val do proc. Jednak w przypadku tego konkretnego wyrażenia, z którym mamy tu do czynienia, operatorem jest symbol f, którego wartość jest obliczana przez operację maszynową `lookup-variable-value` nie zmieniającą wartości żadnych rejestrów. Implementowany przez nas w tym podrozdziale kompilator wykorzysta ten fakt i wygeneruje kod, który oblicza operator za pomocą instrukcji

```
(assign proc (op lookup-variable-value) (const f) (reg env))
```

Kod ten nie tylko nie wykonuje niepotrzebnych operacji na stosie, ale dodatkowo przypisuje znalezioną wartość bezpośrednio do rejestru proc, podczas gdy interpreter najpierw zapamiętałby wynik w val, a następnie przeniósł go do proc.

Kompilator może również optymalizować dostęp do środowiska. Po przeanalizowaniu kodu kompilator w wielu przypadkach potrafi określić, w której

ramce będzie się znajdować dana zmienna, i odwoływać się bezpośrednio do tej ramki, zamiast wyszukiwać ją za pomocą `lookup-variable-value`. Spół implementacji takiego dostępu do zmiennych omówimy w punkcie 5.5.6. Do tego momentu jednak skupimy się na optymalizacjach dotyczących rejestrów i stosu, opisanych powyżej. Istnieje jeszcze wiele innych optymalizacji, które mogą być wykonywane przez kompilator, takich jak „wklejanie” kodu operacji pierwotnych zamiast używania ogólnego mechanizmu `apply` (zob. ćwiczenie 5.38), nie będziemy ich jednak akcentować tutaj. Naszym głównym celem w niniejszym podrozdziale jest zilustrowanie procesu kompilacji w uproszczonym (ale mimo to interesującym) kontekście.

### 5.5.1. Struktura kompilatora

W punkcie 4.1.7 tak zmodyfikowaliśmy nasz początkowy interpreter metacykliczny, aby oddzielić fazę analizy od wykonania. W wyniku analizy każdego wyrażenia powstawała procedura wykonawcza, która otrzymywała jako argument środowisko i wykonywała pożądaną operację. W naszym kompilatorze będziemy zasadniczo wykonywać taką samą analizę. Zamiast jednak tworzyć procedury wykonawcze, będziemy generować ciągi instrukcji przeznaczone do wykonania przez maszynę rejestrową.

Procedura `compile` jest w kompilatorze procedurą rozdzielającą najwyższo poziomu. Odpowiada ona procedurze `eval` z punktu 4.1.1, procedurze `analyze` z punktu 4.1.7 oraz punktowi wejścia `eval-dispatch` z punktu 5.4.1. Kompilator, tak jak interpreter, używa procedur składni wyrażeń, zdefiniowanych w punkcie 4.1.2<sup>35</sup>. Procedura `compile` klasyfikuje kompilowane wyrażenia ze względu na ich składnię. Dla każdego rodzaju wyrażenia wywołuje wyspecjalizowany *generator kodu*:

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp target linkage))
        ((assignment? exp)
         (compile-assignment exp target linkage))
        ((definition? exp)
```

<sup>35</sup> Zwróćmy jednak uwagę, że nasz kompilator jest programem napisanym w języku Scheme i że procedury składniowe, których używa do przetwarzania wyrażeń, są w rzeczywistości procedurami, których używa evaluator metacykliczny. W przeciwnieństwie do tego w przypadku evaluatora z jawnie określonym sterowaniem zakładaliśmy, że równoważne operacje składniowe są dostępne jako operacje maszyny rejestrowej. (Oczywiście, gdy symulowaliśmy maszynę rejestrową w języku Scheme, używaliśmy w symulatorze maszyn rejestrowych tych samych procedur napisanych w języku Scheme).

```

  (compile-definition exp target linkage))
  ((if? exp) (compile-if exp target linkage))
  ((lambda? exp) (compile-lambda exp target linkage))
  ((begin? exp)
    (compile-sequence (begin-actions exp)
      target
      linkage))
  ((cond? exp) (compile (cond->if exp) target linkage))
  ((application? exp)
    (compile-application exp target linkage))
  (else
    (error "Nieznany rodzaj wyrażenia -- COMPILE" exp)))

```

## Wyniki i łączniki

Procedura `compile` i wywoływanie przez nią generatory kodu mają, oprócz kompilowanego wyrażenia, po dwa argumenty. Są to *rejestr docelowy* (ang. *target*), w którym skompilowany kod powinien umieścić wartość wyrażenia, i *deskryptor łącznika* (ang. *linkage descriptor*), który określa, co powinien zrobić kod będący wynikiem kompilacji wyrażenia, gdy zakończy działanie. Deskryptor łącznika może określać, że kod powinien wykonać jedną z następujących trzech rzeczy:

- kontynuować wykonywanie kolejnej instrukcji w ciągu (określamy to za pomocą deskryptora łącznika `next`),
- powrócić z kompilowanej właśnie procedury (określamy to za pomocą deskryptora łącznika `return`),
- wykonać skok do nazwanego punktu wejścia (określamy to za pomocą deskryptora będącego etykietą danego punktu wejścia).

Kompilując na przykład wyrażenie 5 (wyrażenie samoobliczające się) dla rejestru docelowego `val` i łącznika `next`, powinniśmy wygenerować instrukcję

```
(assign val (const 5))
```

Kompilując to samo wyrażenie dla łącznika `return`, powinniśmy wygenerować instrukcję

```
(assign val (const 5))
(goto (reg continue))
```

W pierwszym przypadku jako następna byłaby wykonywana kolejna instrukcja w ciągu. W drugim przypadku nastąpiłby powrót z wywołania procedury. W obu przypadkach wartość wyrażenia została umieszczona w rejestrze docelowym `val`.

## Ciągi instrukcji i wykorzystanie stosu

Wynikiem działania każdego generatora kodu jest *ciąg instrukcji* zawierający kod wynikowy wygenerowany dla danego wyrażenia. Kod generowany dla wyrażeń złożonych powstaje przez złączenie wyników generowania kodu dla prostszych wyrażeń składowych — tak jak obliczanie wyrażeń złożonych jest realizowane poprzez obliczanie wyrażeń składowych.

Najprostsza metoda łączenia ciągów instrukcji jest realizowana przez procedurę `append-instruction-sequences`. Jej argumenty to dowolna liczba ciągów instrukcji, które należy wykonać sekwencyjnie; są one sklejane ze sobą, a wynikiem jest sklejony ciąg. Oznacza to, że jeżeli  $\langle c_1 \rangle$  i  $\langle c_2 \rangle$  są ciągami instrukcji, to wynikiem obliczenia

`(append-instruction-sequences ⟨c12`

jest ciąg

$\langle c_1 \rangle$   
 $\langle c_2 \rangle$

Za każdym razem gdy może zajść konieczność zachowania rejestrów, generatory kodu używają procedury `preserving`, stanowiącej bardziej subtelną metodę łączenia ciągów instrukcji. Procedura ta ma trzy argumenty: zbiór rejestrów i dwa ciągi instrukcji, które należy wykonać sekwencyjnie. Skleja ona te ciągi w taki sposób, że zawartość każdego rejestru ze zbioru jest zachowywana na czas wykonania pierwszego ciągu, jeśli tylko jest ona potrzebna do wykonania drugiego ciągu. Oznacza to, że jeśli pierwszy ciąg modyfikuje dany rejestr, a drugi ciąg faktycznie potrzebuje początkowej wartości tego rejestrów, to `preserving` przed sklejeniem obu ciągów otacza pierwszy z nich instrukcjami `save` i `restore` dla danego rejestrów. W przeciwnym razie wynikiem `preserving` jest po prostu sklejenie dwóch ciągów instrukcji. Na przykład wynikiem

`(preserving (list ⟨r1212`

jest jeden z następujących czterech ciągów instrukcji w zależności od tego, w jaki sposób  $\langle c_1 \rangle$  i  $\langle c_2 \rangle$  używają rejestrów  $\langle r_1 \rangle$  i  $\langle r_2 \rangle$ :

$\langle c_1 \rangle$	<code>(save ⟨r<sub>1</sub></code>	<code>(save ⟨r<sub>2</sub></code>	<code>(save ⟨r<sub>2</sub></code>
$\langle c_2 \rangle$	$\langle c_1 \rangle$	$\langle c_1 \rangle$	$\langle c_1 \rangle$
	<code>(restore ⟨r<sub>1</sub></code>	<code>(restore ⟨r<sub>2</sub></code>	<code>(restore ⟨r<sub>1</sub></code>
	$\langle c_2 \rangle$	$\langle c_2 \rangle$	$\langle c_2 \rangle$

Dzięki łączeniu ciągów instrukcji za pomocą `preserving` kompilator unika niepotrzebnych operacji stosowych. W ten sposób zamykamy również szczegółowo tego, kiedy generować instrukcje `save` i `restore` wewnątrz procedury `preserving`, oddzielając je od kwestii, które powstają przy pisaniu poszczególnych generatorów kodu. W rzeczywistości żadne instrukcje `save` ani `restore` nie są jawnie generowane przez generatorы kodu.

W zasadzie moglibyśmy reprezentować ciągi instrukcji po prostu jako listy instrukcji. Wtedy procedura `append-instruction-sequences` mogłaby łączyć ciągi instrukcji za pomocą zwykłej operacji sklejania list `append`. Wówczas jednak `preserving` byłaby operacją złożoną, ponieważ musiałaby przeanalizować każdy ciąg instrukcji, aby określić, jakie rejestrów są używane w tych ciągach. Procedura `preserving` byłaby nie tylko złożona, ale i nieefektywna, gdyż musiałaby przeanalizować oba ciągi instrukcji będące jej argumentami, mimo że te ciągi mogły same powstać w wyniku wywołania `preserving`, a w takim przypadku ich części byłyby już uprzednio analizowane. Aby uniknąć takich powtarzających się analiz, do każdego ciągu instrukcji dołączymy informację o tym, których rejestrów używa. Gdy będziemy tworzyć podstawowe ciągi instrukcji, podamy tę informację wprost, a procedury łączące ciągi instrukcji będą wyznaczać informacje o używanych rejestrach na podstawie informacji dołączonych do składowych ciągów instrukcji.

Ciąg instrukcji będzie się składał z trzech części:

- zbioru rejestrów, które należy zainicjować przed wykonaniem instrukcji z ciągu (o tych rejestrach mówimy, że są *potrzebne ciągowi*);
- zbioru rejestrów, których wartości są modyfikowane przez instrukcje z ciągu;
- samych instrukcji tworzących ciąg.

Będziemy reprezentować ciąg instrukcji jako listę tych trzech części. Tak więc konstruktor ciągu instrukcji ma postać

```
(define (make-instruction-sequence needs modifies statements)
  (list needs modifies statements))
```

Na przykład ciąg złożony z dwóch instrukcji, które sprawdzają wartość zmiennej `x` w bieżącym środowisku, zapisując wynik w rejestrze `val` i powracając z wywołania procedury, wymaga zainicjowania rejestrów `env` i `continue` oraz modyfikuje rejestr `val`. Zatem taki ciąg konstruujemy jako

```
(make-instruction-sequence '(env continue) '(val)
  '((assign val
        (op lookup-variable-value) (const x) (reg env))
    (goto (reg continue))))
```

Czasami musimy skonstruować pusty ciąg instrukcji:

```
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
```

Procedury służące do łączenia ciągów instrukcji są przedstawione w punkcie 5.5.4.

### Ćwiczenie 5.31

Obliczając zastosowanie procedury, evaluator z jawnie określonym sterowaniem zawsze zachowuje i odtwarza zawartość rejestru env przed i po obliczeniu operatora oraz każdego argumentu (z wyjątkiem ostatniego), zachowuje i odtwarza zawartość rejestru argl przed i po obliczeniu każdego argumentu oraz zachowuje i odtwarza zawartość rejestru proc przed i po obliczeniu ciągu argumentów. Dla każdej z poniższych kombinacji podaj, które z tych operacji save i restore są zbyteczne i mogłyby być wyeliminowane za pomocą mechanizmu zaimplementowanego w kompilatorze przez procedurę preserving:

```
(f 'x 'y)
((f) 'x 'y)
(f (g 'x) y)
(f (g 'x) 'y)
```

### Ćwiczenie 5.32

Z pomocą mechanizmu zaimplementowanego przez procedurę preserving kompilator unika zachowywania i odtwarzania zawartości rejestru env przed i po obliczaniu operatora kombinacji, który jest symbolem. Moglibyśmy wyposażyć w taką optymalizację również evaluator. W rzeczywistości evaluator z jawnie określonym sterowaniem z podrozdziału 5.4 wykonuje już podobną optymalizację, traktując jako osobny przypadek kombinacje bezargumentowe.

- (a) Rozszerz evaluator z jawnie określonym sterowaniem tak, aby rozpoznawał jako osobną klasę wyrażeń kombinacje, których operator to symbol, i wykorzystywał tę informację w trakcie obliczania wyrażeń.
- (b) Liz P. Haker zasugerowała, że rozszerzając evaluator tak, aby rozpoznawał coraz więcej przypadków, moglibyśmy wprowadzić do niego wszystkie optymalizacje stosowane przez kompilator i w ten sposób całkowicie zniwelować przewagę kompilacji. Co sądzisz o tym pomyśle?

### 5.5.2. Kompilowanie wyrażeń

W tym i w następnym punkcie implementujemy generatory kodu, których używa procedura `compile`.

## Kompilowanie kodu łączników

Ogólnie rzecz biorąc, kod generowany przez każdy z generatorów kodu będzie się kończył instrukcjami — generowanymi przez procedurę `compile-linkage` — implementującymi odpowiedni łącznik. Jeśli jest to łącznik `return`, to należy wygenerować instrukcję (`goto (reg continue)`). Instrukcja ta korzysta z rejestru `continue` i nie modyfikuje żadnych rejestrów. Jeśli jest to łącznik `next`, to nie musimy dodawać żadnych instrukcji. W pozostałych przypadkach łącznik jest etykietą i generujemy instrukcję skoku `goto` do tej etykiety — instrukcja ta nie używa ani nie modyfikuje żadnych rejestrów<sup>36</sup>

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
           '((goto (reg continue))))) ((eq? linkage 'next)
         (empty-instruction-sequence)) (else
         (make-instruction-sequence '() '()
           '((goto (label ,linkage)))))))
```

Kod łącznika jest dołączany do ciągu instrukcji przez `preserving` z zachowaniem rejestru `continue`, ponieważ łącznik `return` będzie potrzebował tego rejestru — jeśli dany ciąg instrukcji modyfikuje rejestr `continue`, a kod łącznika potrzebuje go, to `continue` zostanie zachowany i odtworzony.

```
(define (end-with-linkage linkage instruction-sequence)
  (preserving '(continue)
    instruction-sequence
    (compile-linkage linkage)))
```

## Kompilowanie wyrażeń prostych

Generatory kodu dla samoobliczających się wyrażeń, cytowań i zmiennych konstruują ciągi instrukcji, które przypisują rejestrowi docelowemu pożądaną wartość, a następnie kontynuują obliczenia zgodnie z deskryptorem łącznika.

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '() (list target)))
```

<sup>36</sup> W procedurze tej używamy mechanizmu oznaczanego w Lispie *wstecznym apostrofem* (ang. *backquote*; zwany też *odwróconym apostrofem* — ang. *quasiquote*), który jest poręczny przy konstruowaniu list. Poprzedzenie listy wstecznym apostrofem działa w znacznym stopniu tak jak zacytowanie jej, z wyjątkiem tego, że wszystko, co zostało na liście oznaczone przecinkiem, jest obliczane.

Jeśli na przykład wartością `linkage` jest symbol `branch25`, to wartością wyrażenia `'((goto (label ,linkage)))` jest lista `((goto (label branch25)))`. Podobnie, jeżeli wartością `x` jest lista `(a b c)`, to wynikiem `'(1 2 , (car x))` jest lista `(1 2 a)`.

```

  '((assign ,target (const ,exp)))))

(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '() (list target)
      '((assign ,target (const ,(text-of-quotation exp)))))))

(define (compile-variable exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '(env) (list target)
      '((assign ,target
        (op lookup-variable-value)
        (const ,exp)
        (reg env))))))

```

Wszystkie te instrukcje przypisania modyfikują rejestr docelowy, a generator, wyszukując wartość zmiennej, używa rejestrów env.

Przypisania i definicje są realizowane w dużym stopniu tak, jak to miało miejsce w interpreterze. Rekurencyjnie generujemy kod obliczający wartość przypisywaną zmiennej iłączamy do niego ciąg złożony z dwóch instrukcji, który faktycznie ustawia lub definiuje zmienną i zapisuje wartość całego wyrażenia (symbol ok) w rejestrze docelowym. Rejestr docelowy rekurencyjnie wywołanej kompilacji jest val, a dowiązanie to next, tak więc wygenerowany kod umieści swój wynik w rejestrze val i będzie dalej wykonywał dołączony za nim kod. Dołączenie kodu jest wykonywane z zachowaniem zawartości rejestrów env, ponieważ środowisko jest potrzebne przy ustawianiu lub definiowaniu zmiennej, a kod określający wartość zmiennej może być wynikiem kompilacji wyrażenia złożonego, które może w dowolny sposób modyfikować rejesty.

```

(define (compile-assignment exp target linkage)
  (let ((var (assignment-variable exp))
        (get-value-code
          (compile (assignment-value exp) 'val 'next)))
    (end-with-linkage linkage
      (preserving '(env)
        get-value-code
        (make-instruction-sequence '(env val) (list target)
          '((perform (op set-variable-value!)
            (const ,var)
            (reg val)
            (reg env))
            (assign ,target (const ok)))))))

(define (compile-definition exp target linkage)
  (let ((var (definition-variable exp))
        (get-value-code

```

```

  (compile (definition-value exp) 'val 'next)))
  (end-with-linkage linkage
    (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
        '((perform (op define-variable!)
          (const ,var)
          (reg val)
          (reg env))
        (assign ,target (const ok))))))))

```

Dołączany ciąg dwóch instrukcji wymaga użycia rejestrów `env` i `val` oraz modyfikuje rejestr docelowy. Zwróćmy uwagę, że chociaż dla tego ciągu zachowujemy `env`, nie zachowujemy jednak `val`, ponieważ `get-value-code` jest tak skonstruowane, że na potrzeby tego ciągu umieszcza swój wynik wprost w `val`. (W rzeczywistości, gdybyśmy zachowali rejestr `val`, powstałby błąd, gdyż w ten sposób tuż po wykonaniu `get-value-code` zostałaby odtworzona poprzednia wartość `val`).

### Kompilowanie wyrażeń warunkowych

Kod wyrażenia `if` skompilowanego dla danego rejestrów docelowych i łącznika ma postać

```

<skompilowany predykat, rejestr docelowy val, łącznik next>
  (test (op false?) (reg val))
  (branch (label false-branch))
  true-branch
  <następnik skompilowany dla danego rejestr docelowego i łącznika danego lub
  równego after-if>
  false-branch
  <alternatywa skompilowana dla danego rejestr docelowego i łącznika>
  after-if

```

Chcąc wygenerować taki kod, kompilujemy predykat, następnik i altrenatywę, a potemłączymy uzyskany kod z instrukcjami sprawdzającymi wynik predykatu i z dopiero co wygenerowanymi etykietami oznaczającymi człyony kodu odpowiadające prawdziwe i fałszowi oraz koniec wyrażenia warunkowego<sup>37, \*</sup>. Przy takim układzie kodu musimy przeskoczyć kod wygenerowany dla następnika, jeżeli predykat nie jest spełniony. Jedyna drobna komplikacja dotyczy łącznika kończącego kod następnika. Jeśli łącznik całego wyrażenia

<sup>37</sup> Nie możemy po prostu użyć etykiet `true-branch`, `false-branch` i `after-if`, jak to było pokazane wcześniej, gdyż w programie może być więcej wyrażeń `if`. Kompilator używa do generowania etykiet procedury `make-label`. Argumentem tej procedury jest symbol, a wynikiem jest nowy symbol, którego nazwa zaczyna się od nazwy danego symbolu. Na przykład wynikami kolejnych wywołań (`make-label 'a`) będą symbole `a1`, `a2` itd. `Make-label` może być

warunkowego jest równy `return` lub jest etykietą, to zarówno kod następnika, jak i alternatywy będą zakończone takimi samymi łącznikami. Jeśli jednak łącznik całego wyrażenia jest równy `next`, to kod następnika będzie zakończony skokiem, który omija kod alternatywy, do etykiety kończącej kod wyrażenia warunkowego.

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
      (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
              (compile
                (if-consequent exp) target consequent-linkage))
            (a-code
              (compile (if-alternative exp) target linkage)))
        (preserving '(env continue)
          p-code
          (append-instruction-sequences
            (make-instruction-sequence '(val) '()
              '((test (op false?) (reg val))
                (branch (label ,f-branch))))
            (parallel-instruction-sequences
              (append-instruction-sequences t-branch c-code)
              (append-instruction-sequences f-branch a-code)))
          after-if))))))
```

zaimplementowane w następujący sposób, podobny do procedury generującej niepowtarzalne nazwy zmiennych w języku zapytań:

```
(define label-counter 0)

(define (new-label-number)
  (set! label-counter (+ 1 label-counter))
  label-counter)

(define (make-label name)
  (string->symbol
    (string-append (symbol->string name)
      (number->string (new-label-number)))))
```

\* W prawdziwych kompilatorach dla predykatów generowany jest zwykle tzw. „kod skaczący” (ang. *short-circuit*). W wyniku wykonania takiego kodu wartość predykatu nie jest umieszczana w żadnym z rejestrów, natomiast w zależności od tego, czy predykat jest spełniony, czy nie, wykonywany jest skok do jednej z dwóch podanych etykiet. Generowanie takiego kodu przypomina generowanie kodu wyrażeń warunkowych, przy czym dla każdego (pod)wyrażenia określone są dwa łączniki, a nie jeden — co należy zrobić, jeżeli wyrażenie jest prawdziwe, a co, jeżeli jest fałszywe (przyp. tłum.).

Wartość rejestru `env` jest zachowywana na czas wykonania predykatu, ponieważ może być potrzebna w następcu lub alternatywie, a wartość rejestru `continue` jest zachowywana, ponieważ może być potrzebna w łącznikach następcy i alternatywy. Kod następcy i alternatywy (który nie jest wykonywany sekwencyjnie) jest łączony za pomocą specjalnej procedury łączącej `parallel-instruction-sequences` opisanej w punkcie 5.5.4.

Zauważmy, że `cond` jest wyrażeniem pochodnym, więc wszystko, co należy zrobić w celu skompilowania takiego wyrażenia, to zastosować przekształcenie `cond->if` (z punktu 4.1.2) i skompilować otrzymane wyrażenie `if`.

### Kompilowanie ciągów wyrażeń

Kompilując ciągi wyrażeń (pochodzące z treści procedur lub jawnych wyrażeń `begin`), szeregujemy ich wykonywanie. Każde wyrażenie w ciągu jest kompilowane — ostatnie wyrażenie z łącznikiem określonym tak samo jak dla całego wyrażenia, a pozostałe wyrażenia z łącznikami `next` (powodującymi wykonanie pozostały części ciągu). Ciągi instrukcji poszczególnych wyrażeń są sklejane w jeden ciąg instrukcji, przy zachowaniu wartości rejestrów `env` (potrzebnego w pozostały części ciągu) i `continue` (być może potrzebnego w łączniku kończącym cały ciąg).

```
(define (compile-sequence seq target linkage)
  (if (last-exp? seq)
      (compile (first-exp seq) target linkage)
      (preserving '(env continue)
        (compile (first-exp seq) target 'next)
        (compile-sequence (rest-exp seq) target linkage))))
```

### Kompilowanie lambda-wyrażeń

Lambda-wyrażenia tworzą procedury. Kod wynikowy lambda-wyrażenia musi mieć następującą postać:

*(skonstruuj obiekt proceduralny i zapisz go w rejestrze docelowym)  
(łącznik)*

W trakcie kompilacji lambda-wyrażenia generujemy również kod dla treści procedury. Chociaż treść ta nie będzie wykonywana w czasie konstruowania procedury, jednak wygodnie jest umieścić ją w kodzie wynikowym tuż za kodem lambda-wyrażenia. Jeśli łącznik lambda-wyrażenia to `return` lub etykieta, to świetnie. Ale jeśli łącznik jest równy `next`, to musimy przeskoczyć kod treści procedury, stosując łącznik będący etykietą wstawioną po treści procedury. Kod wynikowy ma zatem postać

*(skonstruuj obiekt proceduralny i zapisz go w rejestrze docelowym)  
(kod danego łącznika lub (goto (label after-lambda)))*

*(skompilowana treść procedury)*  
after-lambda

Compile-lambda generuje kod konstruujący obiekt proceduralny, po którym następuje kod treści procedury. W trakcie działania programu obiekt proceduralny zostanie utworzony przez połączenie bieżącego środowiska (środowiska w miejscu definicji) z punktem wejścia skompilowanej treści procedury (dopiero co wygenerowanej etykiety)<sup>38</sup>.

```
(define (compile-lambda exp target linkage)
  (let ((proc-entry (make-label 'entry))
        (after-lambda (make-label 'after-lambda)))
    (let ((lambda-linkage
           (if (eq? linkage 'next) after-lambda linkage)))
      (append-instruction-sequences
       (tack-on-instruction-sequence
        (end-with-linkage lambda-linkage
                           (make-instruction-sequence '(env) (list target)
                                         '(((assign ,target
                                                    (op make-compiled-procedure)
                                                    (label ,proc-entry)
                                                    (reg env))))))
       (compile-lambda-body exp proc-entry)
       after-lambda))))
```

Compile-lambda dołącza kod treści procedury do kodu lambda-wyrażenia nie za pomocą `append-instruction-sequences`, ale za pomocą specjalnej procedury łączącej `tack-on-instruction-sequence` (punkt 5.5.4), ponieważ treść procedury nie stanowi części ciągu instrukcji, który ma być wykonywany razem z całym ciągiem instrukcji, a który jedynie jest umieszczony w tym ciągu, bo tak jest wygodnie.

Compile-lambda-body tworzy kod treści procedury. Kod ten zaczyna się od etykiety oznaczającej punkt wejścia. Dalej następują instrukcje, które powodują przełączenie środowiska wykonywania na odpowiednie środowisko,

<sup>38</sup> Potrzebujemy tutaj instrukcji maszynowych służących do zaimplementowania struktury danych reprezentującej skompilowaną procedurę, analogicznej do struktury reprezentującej procedury złożone, opisanej w punkcie 4.1.3:

```
(define (make-compiled-procedure entry env)
  (list 'compiled-procedure entry env))

(define (compiled-procedure? proc)
  (tagged-list? proc 'compiled-procedure))

(define (compiled-procedure-entry c-proc) (cadr c-proc))

(define (compiled-procedure-env c-proc) (caddr c-proc))
```

w którym ma być obliczana treść procedury — mianowicie środowisko definicji procedury rozszerzone o wiązania przyporządkowujące parametrom formalnym argumenty wywołania procedury. Potem następuje skompilowany ciąg wyrażeń stanowiących treść procedury. Ciąg ten jest kompilowany z łącznikiem `return` i rejestrem docelowym `val`, tak że na koniec nastąpi powrót z procedury, a wynik będzie się znajdował w `val`.

```
(define (compile-lambda-body exp proc-entry)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
      (make-instruction-sequence '(env proc arg1) '(env)
        '(',proc-entry
          (assign env (op compiled-procedure-env) (reg proc))
          (assign env
            (op extend-environment)
            (const ,formals)
            (reg arg1)
            (reg env))))
      (compile-sequence (lambda-body exp) 'val 'return))))
```

### 5.5.3. Kompilowanie kombinacji

Istotę procesu komplikacji stanowi komplikacja zastosowań procedury. Kod kombinacji skompilowany dla danego rejestrów docelowych i łącznika ma postać

*(skompilowany operator dla rejestrów docelowych proc i łącznika next)  
(obliczenie i skonstruowanie listy argumentów w arg1)  
(skompilowane wywołanie procedury dla danego rejestrów docelowych i łącznika)*

Rejestry `env`, `proc` i `arg1` mogą wymagać zachowywania i odtwarzania w trakcie obliczania operatora i argumentów. Zauważmy, że jest to jedynie miejsce w kompilatorze, gdzie określony jest inny rejestr docelowy niż `val`.

Pożądany kod jest generowany przez `compile-application`. Procedura ta rekurencyjnie kompiluje operator w celu utworzenia kodu umieszczającego procedurę, którą należy zastosować, w rejestrze `proc` oraz kompiluje argumenty w celu utworzenia kodu, który oblicza poszczególne argumenty wywołania. Ciągi instrukcji powstałe dla argumentów są łączone (za pomocą `construct-arglist`) z kodem konstruującym listę argumentów w `arg1`. Tak uzyskany kod listy argumentów jest łączony z kodem procedury oraz kodem wywołującym tę procedurę (będącym wynikiem `compile-procedure-call`). Łącząc ciągi instrukcji, musimy zachowywać zawartość tego rejestrów na czas obliczania operatora (ponieważ jego obliczenie może zmodyfikować zawartość rejestrów `env`, a będzie ona potrzebna do obliczenia argumentów) oraz zawartość rejestrów `proc` na czas konstruowania listy argumentów (ponieważ obliczenie tej listy może zmieniać zawartość `proc`, a będzie ona potrzebna do samego zastosowania).

sowania procedury). Zawartość rejestrów `continue` również musi być zachowana przez cały czas, ponieważ jest potrzebna w łączniku wywołania procedury.

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
        (operand-codes
          (map (lambda (operand) (compile operand 'val 'next))
               (operands exp))))
    (preserving '(env continue)
      proc-code
      (preserving '(proc continue)
        (construct-arglist operand-codes)
        (compile-procedure-call target linkage)))))
```

Kod konstruujący listę argumentów będzie obliczał wartość każdego argumentu w `val`, a następnie dołączał ją za pomocą `cons` do listy argumentów gromadzonej w `argl`. Ponieważ gromadzimy argumenty w `argl` po kolejno za pomocą `cons`, musimy zacząć od ostatniego argumentu, a zakończyć na pierwszym, tak aby argumenty pojawiały się na liście wynikowej w kolejności od pierwszego do ostatniego. Zamiast marnować instrukcję, inicjując `argl` jako pustą listę (na potrzeby takiego ciągu obliczeń), pierwszy ciąg instrukcji będzie konstruował początkową wartość `argl`. Tak więc ogólna postać konstrukcji listy argumentów jest następująca:

```
<skompilowane obliczenie ostatniego argumentu w rejestrze docelowym val>
(assign argl (op list) (reg val))
<skompilowane obliczenie poprzedniego argumentu w rejestrze docelowym val>
(assign argl (op cons) (reg val) (reg argl))
...
<skompilowane obliczenie pierwszego argumentu w rejestrze docelowym val>
(assign argl (op cons) (reg val) (reg argl))
```

`argl` musi być zachowywany na czas obliczania każdego argumentu z wyjątkiem pierwszego [chronologicznie; przyp. tłum.] (aby nie utracić zgromadzonych do tej pory argumentów), a `env` musi być zachowywany na czas obliczania każdego z argumentów z wyjątkiem ostatniego [chronologicznie; przyp. tłum.] (na potrzeby obliczania kolejnych argumentów).

Kompilowanie kodu obliczającego argumenty sprawia nieco trudności ze względu na specjalny sposób traktowania argumentu obliczanego jako pierwszy oraz konieczność zachowywania w różnych miejscach zawartości rejestrów `argl` i `env`. Argumentem procedury `construct-arglist` jest lista fragmentów kodu obliczającego poszczególne argumenty. Jeśli wcale nie ma argumentów, to procedura ta generuje tylko instrukcję

```
(assign argl (const ()))
```

W przeciwnym razie `construct-arglist` tworzy kod inicjujący `argl` wartością ostatniego argumentu oraz dołącza kod obliczający pozostałe argumenty i dodający je kolejno do `argl`. Aby przetwarzać argumenty od ostatniego do pierwszego, musimy odwrócić listę fragmentów kodu obliczającego argumenty, dostarczoną przez `compile-application`.

```
(define (construct-arglist operand-codes)
  (let ((operand-codes (reverse operand-codes)))
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl)
          '((assign argl (const ()))))
        (let ((code-to-get-last-arg
              (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence '(val) '(argl)
                  '((assign argl (op list) (reg val)))))))
          (if (null? (cdr operand-codes))
              code-to-get-last-arg
              (preserving '(env)
                code-to-get-last-arg
                (code-to-get-rest-args
                  (cdr operand-codes)))))))
    (if (null? (cdr operand-codes))
        code-to-get-last-arg
        (preserving '(argl)
          (car operand-codes)
          (make-instruction-sequence '(val argl) '(argl)
            '((assign argl
              (op cons) (reg val) (reg argl))))))
        (if (null? (cdr operand-codes))
            code-for-next-arg
            (preserving '(env)
              code-for-next-arg
              (code-to-get-rest-args (cdr operand-codes)))))))
  (define (code-to-get-rest-args operand-codes)
    (let ((code-for-next-arg
          (preserving '(argl)
            (car operand-codes)
            (make-instruction-sequence '(val argl) '(argl)
              '((assign argl
                (op cons) (reg val) (reg argl)))))))
      (if (null? (cdr operand-codes))
          code-for-next-arg
          (preserving '(env)
            code-for-next-arg
            (code-to-get-rest-args (cdr operand-codes)))))))
```

### Stosowanie procedur

Po obliczeniu elementów kombinacji skompilowany kod musi zastosować procedurę pamiętaną w `proc` do argumentów z `argl`. Kod ten dokonuje zasadniczo takiego samego rozdziału co procedura `apply` w evaluatorze metaklicznym (punkt 4.1.1) lub punkt wejścia `apply-dispatch` w evaluatorze z jawnie określonym sterowaniem (punkt 5.4.1). Sprawdza on, czy procedura, którą należy zastosować, jest procedurą pierwotną, czy procedurą skompilowaną. W przypadku procedury pierwotnej jest używana operacja `apply-primitive-procedure`; zobaczymy też wkrótce, w jaki sposób są reali-

zowane procedury skompilowane. Kod stosujący procedurę ma następującą postać:

```

(test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch))
  compiled-branch
  ⟨kod stosujący procedurę skompilowaną dla danego rejestru docelowego
  i odpowiedniego łącznika⟩
  primitive-branch
  (assign ⟨rejestr docelowy⟩
    (op apply-primitive-procedure)
    (reg proc)
    (reg argl))
  ⟨łącznik⟩
  after-call

```

Zauważmy, że człon dla skompilowanej procedury musi po wykonaniu przeskoczyć człon dla procedury pierwotnej. Dlatego też, jeśli początkowym łącznikiem dla wywołania procedury było next, to człon dla procedury złożonej musi użyć łącznika, który przeskoczy do etykiety wstawionej za kodem dla procedury pierwotnej. (Jest to podobne do łącznika zastosowanego w kodzie następnika wyrażenia warunkowego generowanego przez `compile-if`).

```

(define (compile-procedure-call target linkage)
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (after-call (make-label 'after-call)))
    (let ((compiled-linkage
           (if (eq? linkage 'next) after-call linkage)))
      (append-instruction-sequences
        (make-instruction-sequence '(proc) '()
          '((test (op primitive-procedure?) (reg proc))
            (branch (label ,primitive-branch))))
        (parallel-instruction-sequences
          (append-instruction-sequences
            compiled-branch
            (compile-proc-appl target compiled-linkage))
          (append-instruction-sequences
            primitive-branch
            (end-with-linkage linkage
              (make-instruction-sequence '(proc argl)
                (list target)
                '((assign ,target
                  (op apply-primitive-procedure)
                  (reg proc)
                  (reg argl)))))))
        after-call))))

```

Człony kodu odpowiadające procedurom pierwotnym i złożonym, tak jak następnik i alternatywa generowane przez `compile-if`, są łączone za pomocą `parallel-instruction-sequences`, a nie zwykłego `append-instruction-sequences`, ponieważ nie są one wykonywane sekwencyjnie.

### Stosowanie skompilowanych procedur

Kod, który obsługuje zastosowanie procedury, stanowi najbardziej subtelną część kompilatora, mimo że ciąg generowanych przezeń instrukcji jest bardzo krótki. Skompilowana procedura (konstruowana przez `compile-lambda`) ma punkt wejścia, który jest etykietą wyznaczającą początek kodu procedury. Kod w tym punkcie wejścia oblicza wynik w rejestrze `val` i wraca z wywołania, wykonując instrukcję `(goto (reg continue))`. Tak więc możemy oczekiwać, że kod skompilowanego zastosowania procedury (który ma być generowany przez `compile-proc-appl`) dla danego rejestru docelowego i łącznika, powinien wyglądać w następujący sposób, o ile łącznik jest etykietą:

```
(assign continue (label proc-return))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
proc-return
(assign <rejestr docelowy> (reg val)) ; jeśli rejestr docelowy inny niż val
(goto (label <łącznik>)) ; kod łącznika
```

lub też w następujący sposób, jeżeli łącznik jest równy `return`:

```
(save continue)
(assign continue (label proc-return))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
proc-return
(assign <rejestr docelowy> (reg val)) ; jeśli rejestr docelowy inny niż val
	restore continue
(goto (reg continue)) ; kod łącznika
```

Kod ten w taki sposób ustawia `continue`, że powrót z procedury nastąpi do etykiety `proc-return`, po czym wystąpi skok do punktu wejścia procedury. Kod w punkcie `proc-return` przekazuje wynik procedury z `val` do rejestru docelowego (jeśli istnieje taka potrzeba), po czym skacze do miejsca określonego przez łącznik. (Łącznik to zawsze `return` lub etykieta, gdyż `compile-procedure-call` zamienia łączniki `next` w kodzie generowanym dla procedur złożonych na etykiety `after-call`).

W rzeczywistości, jeśli rejestr docelowy jest inny niż `val`, to jest to dokładnie kod generowany przez nasz kompilator<sup>39</sup>. Zwykle jednak rejestr docelowy jest równy `val` (jedyny przypadek, gdy kompilator określa inny rejestr ma miejsce wtedy, gdy chcemy w rejestrze `proc` obliczyć wartość operatora), wynik procedury jest zatem umieszczany bezpośrednio w rejestrze docelowym i nie ma potrzeby powracania do specjalnego miejsca, gdzie wynik jest kopowany. Zamiast tego upraszczamy kod, tak ustawiając `continue`, aby procedura „wracała” bezpośrednio do miejsca określonego przez łącznik wywołującego ją kodu:

```
ustaw continue na łącznik
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

Jeśli łącznik jest etykietą, to tak ustawiamy `continue`, aby procedura, wracając, skoczyła do tej etykiety. (Oznacza to, że ostatnia instrukcja procedury (`goto (reg continue)`) staje się równoważna instrukcji (`goto (label <łącznik>)`) w punkcie `proc-return` powyżej).

```
(assign continue (label <łącznik>))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

Jeśli łącznik jest równy `return`, to nie musimy nic robić z `continue` — rejestr ten ma już pożądaną zawartość. (Oznacza to, że instrukcja (`goto (reg continue)`), która kończy procedurę, skacze bezpośrednio do miejsca, gdzie skoczyłoby (`goto (reg continue)`) w punkcie `proc-return`).

```
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

Przy takiej implementacji łącznika `return` kompilator generuje kod, stosując rekursję ogonową. Wywołanie procedury w ostatnim kroku treści procedury powoduje bezpośrednie przekazanie sterowania bez zapisywania jakiejkolwiek informacji na stosie.

Przypuśćmy, że zamiast tego w celu wywołania procedury z łącznikiem `return` i rejestrzem docelowym `val` wygenerowaliśmy taki kod, jak pokazany powyżej dla rejestrów docelowych innego niż `val`. Zniszczyłoby to rekursję ogonową. Nasz system nadal dawałby takie same wyniki dla dowolnych wyrażeń. Jednak za każdym razem, gdy wywoływalibyśmy procedurę, zachowy-

<sup>39</sup> Tak naprawdę to zgłaszamy błąd, jeżeli rejestr docelowy jest inny niż `val`, a łącznik jest równy `return`, ponieważ łączniki `return` mogą pojawiać się tylko przy kompilowaniu procedur, a nasza konwencja określa, że procedury przekazują wartości w rejestrze `val`.

walibyśmy na stosie zawartość rejestru `continue` i powracalibyśmy z wywołania w celu odtworzenia (niepotrzebnie) zachowanej wartości. Te dodatkowo zapamiętane wartości gromadziłyby się na stosie w trakcie zagnieźdzonych wywołań procedur<sup>40</sup>.

`Compile-proc-appl` generuje powyższy kod zastosowania procedury przez rozważenie czterech przypadków zależnych od tego, czy rejestrem docelowym jest `val` i czy łącznik jest równy `return`. Zwróćmy uwagę na deklaracje mówiące, że ciągi instrukcji mogą modyfikować wszystkie rejestrze; jest tak, ponieważ wykonanie treści procedury może zmieniać zawartość rejestrów w dowolny sposób<sup>41</sup>. Zwróćmy również uwagę, że w przypadku rejestrów docelowych `val` i łącznika `return` zadeklarowano, iż ciąg instrukcji potrzebuje rejestrów `continue` — chociaż `continue` nie jest jawnie używane w dwuelementowym ciągu instrukcji, musimy jednak być pewni, że rejestr ten będzie zawierał poprawną wartość w momencie wejścia do skompilowanej procedury.

```
(define (compile-proc-appl target linkage)
  (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
         (make-instruction-sequence '(proc) all-reg
           '((assign continue (label ,linkage))
             (assign val (op compiled-procedure-entry)
                   (reg proc))
             (goto (reg val)))))

        ((and (not (eq? target 'val))
              (not (eq? linkage 'return)))
         (let ((proc-return (make-label 'proc-return)))
           (make-instruction-sequence '(proc) all-reg
             '((assign continue (label ,proc-return))
               (assign val (op compiled-procedure-entry)
                     (reg proc)))))

        (else (error "Unknown target" target))))
```

<sup>40</sup> Doprzedzenie do tego, aby kompilator generował kod, stosując rekursję ogonową, może wydawać się prostsze. Jednak większość kompilatorów powszechnie stosowanych języków, włączając w to języki C i Pascal, nie stosuje tej techniki i dlatego w tych językach nie można wyrażać procesów iteracyjnych jedynie za pomocą wywołań procedur. Problem z rekursją ogonową w tych językach polega na tym, że ich implementacje używają stosu zarówno do przechowywania argumentów procedur, jak i zmiennych lokalnych oraz adresów powrotu. Opisane w tej książce implementacje języka Scheme przechowują argumenty i zmienne w pamięci, która podlega odśmiecaniu. Powód, dla którego stos jest używany do przechowywania zmiennych i argumentów, jest taki, że unikamy w ten sposób w języku odśmiecania, które w przeciwnym razie byłoby konieczne, a ogólnie jest uważane za mniej efektywne. W rzeczywistości wyszukane kompilatory języka Lisp potrafią używać stosu do przechowywania argumentów bez niszczenia rekursji ogonowej. (Opis tego można znaleźć w [41]). Poza tym toczy się dyskusja, czy przydzielanie pamięci na stosie jest faktycznie bardziej efektywne niż odśmiecanie, jednak padające argumenty zdają się zależeć od szczegółów architektury komputerów. (Przeciwstawne poglądy można znaleźć w [4] oraz [77]).

<sup>41</sup> Zmienna `all-reg` jest związana z listą nazw wszystkich rejestrów:

```
(define all-reg '(env proc val arg1 continue))
```

```

(goto (reg val))
,proc-return
(assign ,target (reg val))
(goto (label ,linkage))))))
((and (eq? target 'val) (eq? linkage 'return))
(make-instruction-sequence '(proc continue) all-reg
'((assign val (op compiled-procedure-entry)
(reg proc))
(goto (reg val))))))
((and (not (eq? target 'val)) (eq? linkage 'return))
(error
"łącznik return, rejestr docelowy różny od val -- COMPILE"
target))))

```

#### 5.5.4. Łączenie ciągów instrukcji

W tym punkcie opisujemy szczegóły reprezentacji i łączenia ciągów instrukcji. Przypomnijmy sobie z punktu 5.5.1, że ciąg instrukcji jest reprezentowany jako lista potrzebnych rejestrów, lista modyfikowanych rejestrów oraz sam ciąg instrukcji. Etykietę (symbol) będziemy również uważać za (zdegenerowany) przypadek ciągu instrukcji, który nie potrzebuje, ani nie modyfikuje żadnych rejestrów. Tak więc do wyznaczania rejestrów potrzebnych i modyfikowanych przez ciągi instrukcji będziemy używać selektorów

```

(define (registers-needed s)
  (if (symbol? s) '() (car s)))

(define (registers-modified s)
  (if (symbol? s) '() (cadr s)))

(define (statements s)
  (if (symbol? s) (list s) (caddr s)))

```

a do określania, czy dany ciąg potrzebuje danego rejestru lub modyfikuje dany rejestr, będziemy używać predykatów

```

(define (needs-register? seq reg)
  (memq reg (registers-needed seq)))

(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))

```

Za pomocą tych predykatów i selektorów możemy zaimplementować różne instrukcje łączące ciągi instrukcji, używane w całym kompilatorze.

Podstawowa operacja łącząca to `append-instruction-sequences`. Jej argumentami są ciągi instrukcji (dowolna ich liczba), które mają być wykonane

sekwencyjnie, a jej wynikiem jest ciąg instrukcji uzyskany przez połączenie wszystkich danych ciągów instrukcji. Delikatną kwestią jest wyznaczenie rejestrów, które są potrzebne, oraz rejestrów, które są modyfikowane przez wynikowy ciąg instrukcji. Ciąg ten modyfikuje te rejesty, które są modyfikowane przez dowolny z danych ciągów; potrzebne są te rejesty, które muszą być zainicjowane przed wykonaniem pierwszego z danych ciągów (tzn. rejesty potrzebne w pierwszym z danych ciągów), oraz te rejesty, które są potrzebne w którymkolwiek z pozostałych ciągów, a które nie są inicjowane (modyfikowane) przez poprzedzające ciągi.

Ciągi są łączone po dwa na raz przez procedurę `append-2-sequences`. Jej argumentami są dwa ciągi instrukcji `seq1` i `seq2`, a wynikiem jest ciąg instrukcji złożony z poleceń z `seq1`, po których następują polecenia z `seq2`. Ciąg ten modyfikuje te rejesty, które modyfikuje `seq1` lub `seq2`, a potrzebuje zawartości tych rejestrów, które są potrzebne w `seq1`, oraz tych, które są potrzebne w `seq2` i nie są modyfikowane przez `seq1`. (Posługując się operacjami na zbiorach, powiemy że nowy zbiór potrzebnych rejestrów jest sumą zbioru rejestrów potrzebnych w `seq1` oraz różnicy zbioru rejestrów potrzebnych w `seq2` i zbioru rejestrów modyfikowanych przez `seq1`). Tak więc procedura `append-instruction-sequences` jest zaimplementowana następująco:

```
(define (append-instruction-sequences . seqs)
  (define (append-2-sequences seq1 seq2)
    (make-instruction-sequence
      (list-union (registers-needed seq1)
                  (list-difference (registers-needed seq2)
                                   (registers-modified seq1)))
      (list-union (registers-modified seq1)
                  (registers-modified seq2)))
    (append (statements seq1) (statements seq2))))
  (define (append-seq-list seqs)
    (if (null? seqs)
        (empty-instruction-sequence)
        (append-2-sequences (car seqs)
                           (append-seq-list (cdr seqs))))))
  (append-seq-list seqs))
```

Procedura ta korzysta z pewnych prostych operacji na zbiorach reprezentowanych w postaci list elementów, podobnych do (nieuporządkowanej) reprezentacji zbiorów opisanej w punkcie 2.3.3:

```
(define (list-union s1 s2)
  (cond ((null? s1) s2)
        ((memq (car s1) s2) (list-union (cdr s1) s2))
        (else (cons (car s1) (list-union (cdr s1) s2)))))
```

```
(define (list-difference s1 s2)
  (cond ((null? s1) '())
        ((memq (car s1) s2) (list-difference (cdr s1) s2))
        (else (cons (car s1)
                     (list-difference (cdr s1) s2))))))
```

Argumentami *preserving*, drugiej co do znaczenia operacji łączącej ciągi instrukcji, są lista rejestrów *regs* oraz dwa ciągi instrukcji *seq1* i *seq2*, które powinny być wykonywane sekwencyjnie. Jej wynikiem jest ciąg instrukcji złożony z poleceń z *seq1*, po których następują polecenia z *seq2*, wraz z odpowiednimi instrukcjami *save* i *restore* otaczającymi *seq1* w celu zachowania tych rejestrów z *regs*, które są modyfikowane przez *seq1*, a są potrzebne w *seq2*. Aby tego dokonać, *preserving* najpierw tworzy ciąg złożony z wymaganych instrukcji *save*, po których następują polecenia z *seq1*, po których z kolei następują wymagane instrukcje *restore*. Taki ciąg potrzebuje, oprócz rejestrów potrzebnych w *seq1*, zawartości zachowywanych i odtwarzanych rejestrów, a modyfikuje te rejesty, które są modyfikowane przez *seq1*, z wyjątkiem rejestrów zachowywanych i odtwarzanych. Potem tak powiększony ciąg oraz ciąg *seq2* są łączone w zwykły sposób. Poniższa procedura implementuje tę technikę rekurencyjnie, przechodząc listę rejestrów, które należy chronić<sup>42</sup>:

```
(define (preserving regs seq1 seq2)
  (if (null? regs)
      (append-instruction-sequences seq1 seq2)
      (let ((first-reg (car regs)))
        (if (and (needs-register? seq2 first-reg)
                 (modifies-register? seq1 first-reg))
            (preserving (cdr regs)
                        (make-instruction-sequence
                          (list-union (list first-reg)
                                      (registers-needed seq1))
                          (list-difference (registers-modified seq1)
                                          (list first-reg)))
                        (append '((save ,first-reg))
                                (statements seq1)
                                '((restore ,first-reg))))
            seq2)
            (preserving (cdr regs) seq1 seq2)))))
```

Kolejna operacja łącząca ciągi, *tack-on-instruction-sequence*, jest używana przez procedurę *compile-lambda* do dołączania treści procedury

<sup>42</sup> Zwróćmy uwagę, że *preserving* wywołuje *append* z trzema argumentami. Mimo że definicja *append* opisana w tej książce dopuszcza tylko dwa argumenty, język Scheme standardowo udostępnia procedurę *append*, której można przekazać dowolną liczbę argumentów.

do innego ciągu. Ponieważ treść procedury nie jest po prostu „wstawką”, którą się wykonuje jako część połączonego ciągu, więc rejestrów, których potrzebuję, nie mają żadnego wpływu na użycie rejestrów przez ciąg, w który jest wbudowana. Dlatego też, dołączając treść procedury do drugiego ciągu, pomijamy zbiory rejestrów potrzebnych i modyfikowanych przez nią.

```
(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
    (registers-needed seq)
    (registers-modified seq)
    (append (statements seq) (statements body-seq))))
```

Procedury `compile-if` i `compile-procedure-call` używają specjalnej operacji łączącej o nazwie `parallel-instruction-sequences` do łączenia dwóch alternatywnych członów obliczeń następujących po sprawdzeniu warunku. Te dwa człony nigdy nie są wykonywane sekwencyjnie; dla każdego konkretnego obliczenia warunku zostanie wybrany tylko jeden z nich. Dlatego też rejestrów potrzebnych w drugim członie są również potrzebne dla połączonego ciągu, nawet jeśli są one modyfikowane w pierwszym członie.

```
(define (parallel-instruction-sequences seq1 seq2)
  (make-instruction-sequence
    (list-union (registers-needed seq1)
                (registers-needed seq2))
    (list-union (registers-modified seq1)
                (registers-modified seq2))
    (append (statements seq1) (statements seq2))))
```

### 5.5.5. Przykład skompilowanego kodu

Teraz, gdy zapoznaliśmy się już ze wszystkimi częściami kompilatora, zbadajmy przykład skompilowanego kodu, aby zobaczyć, jak wszystkie te elementy pasują do siebie. Skompilujemy definicję rekurencyjnej procedury `factorial`, wywołując procedurę `compile`:

```
(compile
'(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
'val
'next)
```

Określiliśmy, że wartość wyrażenia `define` powinna być umieszczana w rejestrze `val`. Nie dbamy o to, co skompilowany kod robi po wykonaniu `define`, więc nasz wybór `next` jako deskryptora łącznika jest arbitralny.

Compile rozpoznaje, że wyrażenie jest definicją, i wywołuje `compile-definition` w celu wygenerowania kodu obliczającego definiowaną wartość (w rejestrze docelowym `val`), po którym następuje kod wprowadzający definicję do środowiska, po którym z kolei następuje kod umieszczający w rejestrze docelowym wartość wyrażenia `define` (która jest symbol `ok`), po którym na koniec następuje kod łącznika. Rejestr `env` jest zachowywany na czas obliczania definiowanej wartości, gdyż jest on potrzebny w momencie wprowadzania definicji do środowiska. Ponieważ łącznik jest równy `next`, więc w tym przypadku nie ma żadnego kodu łącznika. Szkielet skompilowanego kodu ma zatem następującą postać:

```

⟨zachowaj env, jeśli jest modyfikowane przez kod obliczający definiowaną wartość⟩
⟨skompilowana definicja wartości, rejestr docelowy val, łącznik next⟩
⟨odtwórz env, jeśli zostało wcześniej zachowane⟩
(perform (op define-variable!)
  (const factorial)
  (reg val)
  (reg env))
(assign val (const ok))

```

Wyrażenie, które należy skompilować w celu nadania wartości zmiennej `factorial`, to lambda-wyrażenie, którego wartością jest procedura obliczająca silnię. Compile dokonuje tego, wywołując procedurę `compile-lambda`, która kompiluje treść procedury, oznacza ją nowym punktem wejścia, generuje instrukcje łączące treść procedury w nowym punkcie wejścia ze środowiskiem wykonania i zapisujące wynik do rejestru `val`. Ciąg instrukcji przeskakuje następnie wstawiony w tym miejscu skompilowany kod. Sam kod procedury na początek rozszerza środowisko, w którym procedura była zdefiniowana, o ramkę wiążącą parametr formalny `n` z argumentem procedury. Dalej następuje sama treść procedury. Ponieważ kod obliczający wartość zmiennej nie zmienia zawartości rejestru `env`, nie ma potrzeby generowania opcjonalnych instrukcji `save` i `restore` pokazanych powyżej. (Kod procedury w punkcie wejścia `entry2` nie jest wykonywany w tym momencie, więc kwestia użycia `env` nie ma znaczenia). Zatem szkielet skompilowanego kodu ma postać

```

(assign val (op make-compiled-procedure)
  (label entry2)
  (reg env))
(goto (label after-lambda1))
entry2
(assign env (op compiled-procedure-env) (reg proc))

```

```

(assign env (op extend-environment)
       (const (n))
       (reg arg1)
       (reg env))
<skompilowana treść procedury>
after-lambda1
(perform (op define-variable!)
          (const factorial)
          (reg val)
          (reg env))
(assign val (const ok))

```

Treść procedury jest zawsze kompilowana (przez `compile-lambda-body`) dla rejestru docelowego `val` i łącznika `return`. W tym przypadku kompilowany ciąg składa się z jednego wyrażenia `if`:

```

(if (= n 1)
    1
    (* (factorial (- n 1)) n))

```

Procedura `compile-if` generuje kod, który najpierw oblicza predykat (dla rejestru docelowego `val`), a następnie sprawdza wynik i przeskakuje następnik, jeżeli predykat nie jest spełniony. Zawartości rejestrów `env` i `continue` są zachowywane na czas obliczania predykatu, gdyż mogą być potrzebne w pozostałej części wyrażenia `if`. Ponieważ wyrażenie `if` jest ostatnim (i jedynym) wyrażeniem w ciągu tworzącym treść procedury, zatem jego rejestrem docelowym jest `val`, a łącznik jest równy `return`, tak więc zarówno następnik, jak i alternatywa są kompilowane dla rejestru docelowego `val` i łącznika `return`. (Oznacza to, że wartość instrukcji warunkowej, która jest obliczana przez następnik lub alternatywę, jest wartością procedury).

```

<zachowaj continue i env, jeśli są modyfikowane przez predykat, następnik lub alternatywę>
<skompilowany predykat, rejestr docelowy val, łącznik next>
<odtwórz continue i env, jeśli zostały zachowane powyżej>
(test (op false?) (reg val))
(branch (label false-branch4))
true-branch5
<skompilowany następnik, rejestr docelowy val, łącznik return>
false-branch4
<skompilowana alternatywa, rejestr docelowy val, łącznik return>
after-if3

```

Predykat `(= n 1)` to wywołanie procedury. Odnajduje on wartość operatora (symbol `=`) i umieszcza ją w rejestrze `proc`. Następnie sprawdza, czy `proc` zawiera operację pierwotną, czy procedurę złożoną, i skacze odpowied-

nio do członu przetwarzającego operacje pierwotne lub procedury złożone. Oba człony kontynuują dalsze obliczenia od etykiety `after-call`. Wymagania dotyczące zachowywania zawartości rejestrów na czas obliczania operatora i argumentów nie wymuszają zachowywania zawartości żadnych rejestrów, ponieważ rejesty, których mogłoby to dotyczyć, nie są modyfikowane.

```

(assign proc
        (op lookup-variable-value) (const =) (reg env))
(assign val (const 1))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value) (const n) (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch17))

compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))

primitive-branch17
  (assign val (op apply-primitive-procedure)
        (reg proc)
        (reg argl))

after-call15

```

Następnik instrukcji warunkowej, który jest stałą 1, po skompilowaniu (dla rejestrów docelowego `val` i łącznika `return`) ma postać

```

(assign val (const 1))
(goto (reg continue))

```

Kod dla alternatywy to kolejne wywołanie procedury, gdzie procedura ta jest określona przez symbol `*`, a argumentami są `n` i wynik kolejnego wywołania procedury (`factorial`). Każde z tych wywołań ustawia rejesty `proc` i `argl` oraz zawiera własne człony wywołujące procedury pierwotne i złożone. Na rysunku 5.17 jest pokazany pełny skompilowany kod definicji procedury `factorial`. Zauważmy, że potencjalne instrukcje `save` i `restore` dotyczące rejestrów `continue` i `env`, umieszczone wokół predykatu, są rzeczywiście generowane, ponieważ zawartości tych rejestrów są modyfikowane przez wywołanie procedury w predykatie, a są potrzebne do wywołania procedury oraz w łączniku `return` w członach wyrażenia warunkowego.

**Ćwiczenie 5.33**

Rozważ następującą definicję procedury obliczającej silnię, nieznacznie różniącą się od tej podanej na początku tego punktu:

```
(define (factorial-alt n)
  (if (= n 1)
    1
    (* n (factorial-alt (- n 1)))))
```

Skompiluj tę procedurę i porównaj otrzymany kod z wygenerowanym dla procedury `factorial`. Wyjaśnij różnice, które zauważysz. Czy któryś z tych programów jest wykonywany bardziej efektywnie niż drugi?

**Ćwiczenie 5.34**

Skompiluj iteracyjną procedurę obliczającą silnię:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Uzupełnij wynikowy kod komentarzami wskazującymi zasadnicze różnice między kodem iteracyjnej i rekurencyjnej wersji `factorial`, które sprawiają, że jeden proces gromadzi informacje na stosie, a drugi działa w stałej pamięci.

**Ćwiczenie 5.35**

Jakie wyrażenie po skompilowaniu daje kod przedstawiony na rys. 5.18?

**Ćwiczenie 5.36**

Jaką kolejność obliczania stosuje nasz kompilator wobec argumentów kombinacji? Czy jest to kolejność od lewej do prawej, od prawej do lewej, czy jeszcze jakaś inna? Gdzie w kompilatorze jest określona ta kolejność? Zmodyfikuj tak kompilator, aby stosował jakąś inną kolejność. (Zobacz omówienie kolejności obliczania w evaluatorze z jawnie określonym sterowaniem w punkcie 5.4.1). W jaki sposób zmiana kolejności obliczania argumentów wpływa na wydajność kodu budującego listę argumentów?

**Ćwiczenie 5.37**

Jeden ze sposobów umożliwiających zrozumienie działania mechanizmu `preserving` optymalizującego w kompilatorze odwołania do stosu polega na stwierdzeniu, jakie dodatkowe operacje byłyby wygenerowane, gdyby ta technika nie była zastosowana. Zmodyfikuj procedurę `preserving` tak, aby zawsze generowała operacje `save` i `restore`. Skompiluj jakieś proste wyrażenie i znajdź w nim niepotrzebne operacje stosowe, które zostały wygenerowane. Porównaj uzyskany kod z kodem wygenerowanym przy nienaruszonym mechanizmie `preserving`.

```

;; zbuduj procedurę i przeskocz jej skompilowaną treść
(assign val
        (op make-compiled-procedure) (label entry2) (reg env))
(goto (label after-lambda1))

entry2      ; wywołania factorial będą skakać tutaj
(assign env (op compiled-procedure-env) (reg proc))
(assign env
        (op extend-environment) (const (n)) (reg argl) (reg env))
;; początek właściwej treści procedury
(save continue)
(save env)

;; oblicz (= n 1)
(assign proc (op lookup-variable-value) (const =) (reg env))
(assign val (const 1))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value) (const n) (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch17))

compiled-branch16
(assign continue (label after-call15))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))

primitive-branch17
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call15 ; val zawiera teraz wynik (= n 1)
	restore env
	restore continue
(test (op false?) (reg val))
(branch (label false-branch4))

true-branch5 ; przekaż w wyniku 1
(assign val (const 1))
(goto (reg continue))

false-branch4
;; oblicz i przekaż w wyniku (* (factorial (- n 1)) n)
(assign proc (op lookup-variable-value) (const *) (reg env))
(save continue)
(save proc) ; zachowaj procedurę *
(assign val (op lookup-variable-value) (const n) (reg env))
(assign argl (op list) (reg val))
(save argl) ; zachowaj częściową listę argumentów *

;; oblicz (factorial (- n 1)), co stanowi drugi argument *
(assign proc
        (op lookup-variable-value) (const factorial) (reg env))
(save proc) ; zachowaj procedurę factorial

```

Rys. 5.17. Skompilowana definicja procedury factorial (ciąg dalszy na następnej stronie)

```

;; oblicz (- n 1), co stanowi argument factorial
  (assign proc (op lookup-variable-value) (const -) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch8))
compiled-branch7
  (assign continue (label after-call16))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch8
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call16 ; val zawiera teraz wynik (- n 1)
  (assign argl (op list) (reg val))
  (restore proc) ; odtwórz factorial
;; zastosuj factorial
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch11))
compiled-branch10
  (assign continue (label after-call19))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch11
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call19 ; val zawiera teraz wynik (factorial (- n 1))
  (restore argl) ; odtwórz częściową listę argumentów *
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc) ; odtwórz *
  (restore continue)
;; zastosuj * i przekaż jej wynik
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch14))
compiled-branch13
;; zwróćmy uwagę na rekursję ogonową przy wywołaniu procedury złożonej
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch14
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call12
after-if3
after-lambda1
;; przypisz procedurę zmiennej factorial
  (perform
    (op define-variable!) (const factorial) (reg val) (reg env))
  (assign val (const ok))


```

Rys. 5.17. (ciąg dalszy)

```

(assign val (op make-compiled-procedure) (label entry16)
       (reg env))
(goto (label after-lambda15))
entry16
(assign env (op compiled-procedure-env) (reg proc))
(assign env
       (op extend-environment) (const (x)) (reg argl) (reg env))
(assign proc (op lookup-variable-value) (const +) (reg env))
(save continue)
(save proc)
(save env)
(assign proc (op lookup-variable-value) (const g) (reg env))
(save proc)
(assign proc (op lookup-variable-value) (const +) (reg env))
(assign val (const 2))
(assign argl (op list) (reg val))
(assign val (op lookup-variable-value) (const x) (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch19))
compiled-branch18
(assign continue (label after-call17))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
primitive-branch19
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call17
(assign argl (op list) (reg val))
	restore proc
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch22))
compiled-branch21
(assign continue (label after-call20))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
primitive-branch22
(assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call20
(assign argl (op list) (reg val))
	restore env
(assign val (op lookup-variable-value) (const x) (reg env))
(assign argl (op cons) (reg val) (reg argl))
	restore proc
	restore continue
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch25))

```

Rys. 5.18. Przykład kodu wynikowego (ciąg dalszy na następnej stronie). Zobacz Ćwiczenie 5.35

```

compiled-branch24
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch25
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call123
after-lambda15
  (perform (op define-variable!) (const f) (reg val) (reg env))
  (assign val (const ok))

```

Rys. 5.18. (ciąg dalszy)

### Ćwiczenie 5.38

Nasz kompilator jest na tyle sprytny, że unika niepotrzebnych operacji stosowych. Jednak wcale nie wykazuje sprytnego w trakcie kompilowania wywołań procedur pierwotnych języka do operacji pierwotnych udostępnianych przez maszynę. Rozważmy na przykład, ile kodu jest generowanego w celu obliczenia ( $+ a 1$ ): Kod ten najpierw umieszcza listę argumentów w `argl`, w `proc` umieszcza pierwotną procedurę dodawania (która znajduje, sprawdzając w środowisku znaczenie symbolu `+`) oraz sprawdza, czy procedura ta jest pierwotna, czy też złożona. Kompilator zawsze generuje kod sprawdzający ten warunek, jak również kod członów odpowiadających procedurom pierwotnej i złożonej (a tylko jeden z nich jest wykonywany). Nie pokazaliśmy części sterownika implementującej operacje pierwotne, przyjmujemy jednak, że instrukcje te korzystają z pierwotnych operacji arytmetycznych zrealizowanych przez ścieżki danych maszyny. Zastanów się, o ile mniej kodu byłoby generowane, gdyby kompilator mógł kodować operacje, używając *kodu otwartego* — tzn. gdyby mógł generować kod bezpośrednio wykorzystujący pierwotne operacje maszynowe. Wyrażenie ( $+ a 1$ ) mogłoby być wówczas skompilowane do tak prostego kodu, jak<sup>43</sup>

```

(assign val (op lookup-variable-value) (const a) (reg env))
(assign val (op +) (reg val) (const 1))

```

W ćwiczeniu tym rozszerzymy nasz kompilator o otwarte kodowanie wybranych operacji pierwotnych. Dla wywołań tych procedur pierwotnych zamiast ogólnego kodu stosującego procedury będzie generowany wyspecjalizowany kod. Aby to zrealizować, rozszerzymy naszą maszynę o dwa specjalne rejesty przeznaczone na argumenty, `arg1` i `arg2`. Pierwotne operacje arytmetyczne maszyny będą pobierać dane z `arg1` i `arg2`, a wyniki będą mogły być umieszczane w `val`, `arg1` lub `arg2`.

Kompilator musi być w stanie rozpoznawać w kodzie źródłowym bezpośrednie zastosowania operacji pierwotnych. Rozszerzymy w procedurze `compile` analizę przypadków, aby oprócz aktualnie rozpoznawanych słów kluczowych (form specjalnych)

<sup>43</sup> Użyliśmy tutaj tego samego symbolu `+` na oznaczenie zarówno procedur z języka źródłowego, jak i operacji maszynowych. W ogólności nie będzie odpowiedniości jeden do jednego między operacjami pierwotnymi języka źródłowego i operacjami pierwotnymi maszyny.

były również rozpoznawane nasze operacje pierwotne<sup>44</sup>. Dla każdej formy specjalnej nasz kompilator zawiera generator kodu. W tym ćwiczeniu będziemy konstruować rodzinę generatorów kodu otwartego dla operacji pierwotnych.

- (a) Wszystkie operacje pierwotne o otwartym kodzie, w odróżnieniu od form specjalnych, wymagają obliczenia swoich argumentów. Napisz generator kodu `spread-arguments`, którego będą używać wszystkie generatory kodu otwartego. Argumentem `spread-arguments` powinna być lista argumentów. Procedura ta powinna generować kod obliczający dane argumenty i umieszczający ich wyniki w kolejnych rejestrach argumentów. Zauważ, że argument może zawierać wywołanie operacji pierwotnej kompilowanej do kodu otwartego, a wtedy rejesty argumentów muszą być zachowywane na czas obliczania argumentu.
- (b) Dla każdej z procedur pierwotnych `=`, `*`, `-` i `+` napisz generator kodu, którego argumentami są kombinacja z takim właśnie operatorem oraz rejestr docelowy i deskryptor łącznika. Ma on generować kod obliczający argumenty w odpowiednich rejestrach, a następnie wykonywać określona operację dla danego rejestru docelowego i łącznika. Obsługiwane mają być tylko wyrażenia dwuargumentowe. Zmień procedurę `compile` tak, aby przekazywała odpowiednie przypadki do generatorów kodu.
- (c) Wypróbuj zmodyfikowany kompilator na przykładzie procedury `factorial`. Porównaj powstały kod z kodem otrzymanym bez kodowania otwartego.
- (d) Rozszerz generatory dla `+` i `*` tak, aby obsługiwały wyrażenia o dowolnej liczbie argumentów. Wyrażenie o więcej niż dwóch argumentach będzie musiało być skompilowane jako ciąg operacji, z których każda ma tylko dwa argumenty.

### 5.5.6. Adresowanie składniowe

Jedna z najczęściej stosowanych przez kompilatory optymalizacji dotyczy wyszukiwania zmiennych w środowisku. Nasz kompilator, tak jak go do tej pory zaimplementowaliśmy, generuje kod, który używa operacji `lookup-variable-value` maszyny evaluatora. Operacja ta szuka zmiennej, porównując jej nazwę z każdą zmienną, która jest aktualnie związana, i przeglądając kolejne ramki tworzące środowisko wykonania. Takie przeszukiwanie może być kosztowne, jeśli ramki są głęboko pozagieźdzane lub jeśli jest wiele zmiennych. Rozważmy na przykład problem wyszukania wartości `x` w trakcie obliczania wyrażenia `(* x y z)` w zastosowaniu procedury będącej wynikiem

```
(let ((x 3) (y 4))
  (lambda (a b c d e)
    (let ((y (* a b x)))
      (z (+ c d x)))
    (* x y z))))
```

<sup>44</sup> Zamiana operacji pierwotnych w słowa kluczowe jest, ogólnie rzecz biorąc, złym pomysłem, gdyż użytkownik nie może wówczas związać nazw tych operacji z jakimś innymi procedurami. Ponadto, jeżeli dodamy słowa kluczowe do działającego kompilatora, to istniejące programy, które definiują procedury o takich samych nazwach, przestaną działać. Wskazówki, jak można tego uniknąć, zawiera ćwiczenie 5.44.

Ponieważ wyrażenie  $\lambda e$  jest tylko lukrem syntaktycznym pokrywającym kombinację lambda-wyrażenia, wyrażenie to jest równoważne

```
((lambda (x y)
  (lambda (a b c d e)
    ((lambda (y z) (* x y z))
     (* a b x)
     (+ c d x))))
 3
 4)
```

Za każdym razem, gdy `lookup-variable-value` sięga do  $x$ , musi stwierdzić, że symbol  $x$  nie jest równy (w sensie `eq?`) ani  $z$  (w pierwszej ramce), ani też  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$  (w drugiej ramce). Założymy na chwilę, że nasz program nie używa `define` — czyli że zmienne są wiązane tylko za pomocą lambda-wyrażeń. Ponieważ w naszym języku obowiązują wiązania składniowe, środowisko wykonania dowolnego wyrażenia będzie miało strukturę oddającą składniową strukturę programu, w którym występuje dane wyrażenie<sup>45</sup>. Tak więc kompilator może wywnioskować, analizując powyższe wyrażenie, że za każdym razem, gdy procedura jest stosowana, zmienna  $x$  w  $(* x y z)$  będzie pierwszą zmienną w trzeciej ramce, licząc od bieżącej ramki.

Możemy wykorzystać ten fakt, wprowadzając nowy rodzaj operacji odnajdującej zmienne, `lexical-address-lookup`, której argumentami są środowisko oraz *adres składniowy* zawierający dwie liczby: *numer ramki* określający, ile ramek należy pominąć, oraz *przesunięcie*, które określa, ile zmiennych należy pominąć w obrębie ramki. Wynikiem `lexical-address-lookup` jest wartość zmiennej przechowywanej pod danym adresem składniowym w bieżącym środowisku. Jeśli dodamy do naszej maszyny operację `lexical-address-lookup`, to możemy generować za pomocą kompilatora kod, który odwołuje się do zmiennych za pomocą tej operacji, a nie za pomocą `lookup-variable-value`. Podobnie nasz skompilowany kod może używać nowej operacji `lexical-address-set!` zamiast `set-variable-value!`.

Aby wygenerować taki kod, kompilator musi być w stanie określić adres składniowy zmiennej, do której odwołanie ma skompilować. Adres składniowy zmiennej w programie zależy od tego, gdzie znajduje się on w kodzie. Na przykład w następującym programie adres zmiennej  $x$  w wyrażeniu  $\langle e1 \rangle$  wynosi  $(2,0)$  — dwie ramki w dół, pierwsza zmienna w ramce. W tym samym miejscu zmienna  $y$  ma adres  $(0,0)$ , a  $c$  adres  $(1,2)$ . W wyrażeniu  $\langle e2 \rangle$  zmienna  $x$  ma adres  $(1,0)$ ,  $y$  ma adres  $(1,1)$ , a  $c$  ma adres  $(0,2)$ .

<sup>45</sup> Stwierdzenie to przestaje być prawdziwe, jeżeli dopuścimy definicje wewnętrzne, chyba że zostaną one wyłuskane. Zobacz ćwiczenie 5.43.

```
((lambda (x y)
  (lambda (a b c d e)
    ((lambda (y z) (e1))
     (e2)
     (+ c d x))))
 3
 4)
```

Jeden ze sposobów, w jaki kompilator może generować kod korzystający z adresowania składniowego, polega na utrzymywaniu struktury danych nazywanej *środowiskiem kompilacji* (ang. *compile-time environment*). Struktura ta śledzi, które zmienne będą na jakich pozycjach i w których ramkach w środowisku wykonania, w momencie wykonania określonej operacji dotyczącej zmiennych. Środowisko kompilacji to lista ramek, z których każda zawiera listę zmiennych. (Oczywiście ze zmiennymi nie są wiązane żadne wartości, ponieważ nie są one obliczane w trakcie kompilacji). Środowisko kompilacji staje się dodatkowym argumentem `compile` i jest przekazywane dalej do każdego generatora kodu. Wywołanie `compile` najwyższego poziomu używa pustego środowiska kompilacji. Gdy kompilowana jest treść lambda-wyrażenia, `compile-lambda-body` rozszerza środowisko kompilacji o ramkę zawierającą parametry procedury, tak że ciąg wyrażeń tworzących treść procedury jest kompilowany w rozszerzonym środowisku. W trakcie całej kompilacji `compile-variable` i `compile-assignment` wykorzystują środowisko kompilacji do generowania odpowiednich adresów składniowych.

W ćwiczeniach 5.39–5.43 jest opisane, jak można uzupełnić tak naszkicowaną technikę adresowania składniowego w celu wprowadzenia do kompilatora składniowego wyszukiwania zmiennych. Ćwiczenie 5.44 dotyczy innego zastosowania środowiska kompilacji.

### Ćwiczenie 5.39

Napisz procedurę `lexical-address-lookup` implementującą nową operację wyszukiwania zmiennych. Powinna ona mieć dwa argumenty: adres składniowy i środowisko wykonania, a jej wynikiem powinna być wartość zmiennej pamiętana pod podanym adresem składniowym. `Lexical-address-lookup` powinna zgłaszać błąd, jeśli adres zmiennej jest symbolem `*unassigned*`<sup>46</sup>. Napisz również procedurę `lexical-address-set!` implementującą operację zmiany wartości zmiennej o określonym adresie składniowym.

### Ćwiczenie 5.40

Zmodyfikuj kompilator tak, aby utrzymywał w opisany powyżej sposób środowisko kompilacji. To znaczy, dodaj do procedury `compile` i rozmaitych generato-

<sup>46</sup> Jest to modyfikacja wyszukiwania wartości zmiennej, które jest potrzebne do zaimplementowania eliminacji definicji wewnętrznych metodą ich wyłuskiwania (ćwiczenie 5.43). Aby działało adresowanie składniowe, trzeba wyeliminować te definicje.

rów kodu argument reprezentujący środowisko kompilacji i rozszerz je w `compile-lambda-body`.

### Ćwiczenie 5.41

Napisz procedurę `find-variable`, której argumentami są zmienna i środowisko kompilacji, a wynikiem jest adres składniowy tej zmiennej w danym środowisku. Na przykład w pokazanym na poprzedniej stronie fragmencie kodu, w trakcie kompilacji wyrażenia `(e1)` środowisko kompilacji jest postaci `((y z) (a b c d e) (x y))`. `Find-variable` powinno dawać następujące wyniki:

```
(find-variable 'c '((y z) (a b c d e) (x y)))
(1 2)

(find-variable 'x '((y z) (a b c d e) (x y)))
(2 0)

(find-variable 'w '((y z) (a b c d e) (x y)))
not-found
```

### Ćwiczenie 5.42

Używając procedury `find-variable` z ćwiczenia 5.41, tak zmodyfikuj procedury `compile-variable` i `compile-assignment`, aby generowały instrukcje wykorzystujące adresy składniowe. W przypadku gdy wynikiem `find-variable` jest `not-found` (tzn. gdy zmienna nie występuje w środowisku kompilacji), generatory kodu, tak jak poprzednio, powinny odnajdować wiązanie za pomocą operacji ewaluatora. (Jedyne miejsce, gdzie może znajdować się zmienna, której nie odnaleziono w trakcie kompilacji, to środowisko globalne, które jest częścią środowiska wykonywania, ale nie jest częścią środowiska kompilacji<sup>47</sup>). Tak więc, jeśli chcesz, operacje ewaluatora mogą zaglądać bezpośrednio do środowiska globalnego, które można uzyskać za pomocą operacji (`op get-global-environment`), zamiast wyszukiwać ich w całym środowisku pamiętanym w `env`). Przetestuj zmodyfikowany kompilator na kilku prostych przykładach, takich jak zagnieździone kombinacje lambda-wyrażeń przedstawione na początku niniejszego punktu.

### Ćwiczenie 5.43

W punkcie 4.1.6 przekonywaliśmy, że w strukturze blokowej definicje wewnętrzne nie powinny być traktowane jak „prawdziwe” `define`. Zamiast tego treść procedury powinna być interpretowana tak, jakby definiowane zmienne wewnętrzne były zainstalowane jako zwykłe zmienne `lambda` zainicjowane ich poprawnymi wartościami za pomocą `set!`. W punkcie 4.1.6 i ćwiczeniu 4.16 pokazaliśmy, jak należy w tym celu zmodyfikować interpreter metacykliczny, wyłuskując definicje wewnętrzne. Zmodyfi-

<sup>47</sup> Adresy składniowe nie mogą być używane do uzyskiwania dostępu do zmiennych w środowisku globalnym, ponieważ mogą one być definiowane i przedefiniowywane interakcyjnie w dowolnym momencie. Po wyłuskaniu definicji lokalnych, jak opisano to w ćwiczeniu 5.43, jedynie definicje, jakie widzi kompilator, to te z najwyższego poziomu, które dotyczą środowiska globalnego. Kompilacja definicji nie powoduje wstawienia definiowanej nazwy do środowiska kompilacji.

kuj kompilator tak, aby wykonywał takie samo przekształcenie przed skompilowaniem treści procedury.

### Ćwiczenie 5.44

W niniejszym punkcie skupiliśmy się na zastosowaniu środowiska kompilacji w celu uzyskania adresów składowych. Istnieją jednak inne zastosowania środowiska kompilacji. Na przykład w ćwiczeniu 5.38 zwiększyliśmy efektywność generowanego kodu, stosując kodowanie otwarte operacji pierwotnych. W naszej implementacji nazwy procedur kodowanych otwarcie były kodowane jak słowa kluczowe. Gdyby program miał zmieniać wiązania takich nazw, mechanizm opisany w ćwiczeniu 5.38 nadal kodowałby je otwarcie jako operacje pierwotne, pomijając nowe wiązanie. Rozważmy na przykład procedurę

```
(lambda (+ * a b x y)
  (+ (* a x) (* b y)))
```

która oblicza liniową kombinację  $x$  i  $y$ . Moglibyśmy ją wywoływać z argumentami `+matrix`, `*matrix` i czterema macierzami, ale kompilator z kodowaniem otwartym nadal generowałby w wyrażeniu  $(+ (* a x) (* b y))$  dla operacji `+ i *` kod otwarty operacji pierwotnych `+ i *`. Zmodyfikuj kompilator z kodowaniem otwartym tak, aby sprawdzał środowisko kompilacji po to, żeby móc generować poprawny kod dla wyrażeń zawierających nazwy procedur pierwotnych. (Generowany kod będzie działał poprawnie, pod warunkiem że program nie będzie wykonywał dla tych nazw operacji `define` ani `set!`).

#### 5.5.7. Scalanie skompilowanego kodu i ewaluatora

Nie wyjaśniliśmy jeszcze, jak wprowadzać skompilowany kod do maszyny ewaluatora i jak go uruchamiać. Zakładamy, że ewaluator z jawnie określonym sterowaniem został zdefiniowany tak jak w punkcie 5.4.4 wraz z dodatkowymi operacjami określonymi w przypisie 38. Zaimplementujemy procedurę `compile-and-go`, która kompiluje wyrażenia języka Scheme, wprowadza powstały kod wynikowy do maszyny ewaluatora i powoduje wykonanie kodu przez tę maszynę w środowisku globalnym ewaluatora, wypisanie wyników i powrót do pętli sterującej ewaluatora. Zmodyfikujmy również ewaluator w ten sposób, że interpretowane wyrażenia będą mogły wywoływać zarówno skompilowane, jak i interpretowane procedury. Możemy wówczas wprowadzić skompilowaną procedurę do maszyny i do jej wywoływania używać ewaluatora:

```
(compile-and-go
  '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
```

---

```
;;; EC-Eval wartość:
ok

;;; EC-Eval wprowadź wyrażenie:
(factorial 5)
;;; EC-Eval wartość:
120
```

W celu umożliwienia ewaluatorowi wywoływania skompilowanych procedur (aby np. obliczyć powyższe wywołanie `factorial`) musimy tak zmienić kod `apply-dispatch` (punkt 5.4.1), aby rozpoznawał procedury skompilowane (jako inny rodzaj procedur niż złożone i pierwotne) i przekazywał sterowanie bezpośrednio do punktu wejścia skompilowanego kodu<sup>48</sup>:

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (test (op compiled-procedure?) (reg proc))
  (branch (label compiled-apply))
  (goto (label unknown-procedure-type))

compiled-apply
  (restore continue)
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
```

Zwróćmy uwagę na odtworzenie `continue` w `compiled-apply`. Przypomnijmy, iż ewaluator jest tak zorganizowany, że w `apply-dispatch` adres powrotu znajduje się na wierzchołku stosu. Jednakże skompilowany kod oczekuje, że w punkcie wejścia adres powrotu znajduje się w rejestrze `continue`, dlatego rejestr ten musi być odtworzony przed wykonaniem skompilowanego kodu.

Aby umożliwić nam uruchomienie jakiegoś skompilowanego kodu w momencie uruchomienia maszyny ewaluatora, dodajemy na początku maszyny ewaluatora instrukcję `branch`, która wykonuje skok do nowego punktu wejścia, o ile jest ustawiony rejestr `flag`<sup>49</sup>.

---

<sup>48</sup> Oczywiście zarówno procedury skompilowane, jak i interpretowane są złożone (nie są pierwotne). W celu zachowania zgodności z terminologią używaną w ewaluatorze z jawnie określonym sterowaniem w niniejszym punkcie będziemy używać określenia „złożona” w znaczeniu „interpretowana” (w przeciwnieństwie do skompilowanej).

<sup>49</sup> Teraz, gdy maszyna ewaluatora wykonuje najpierw instrukcję `branch`, zawsze przed jej uruchomieniem musimy inicjować rejestr `flag`. Chcąc uruchomić maszynę tak, żeby wykonywała zwykłą pętlę wczytaj-oblicz-wypisz, możemy użyć

---

```
(branch (label external-entry))      ; skacze, jeśli flag jest ustawiony
read-eval-print-loop
  (perform (op initialize-stack))
  ...

```

External-entry zakłada, że maszyna została uruchomiona z rejestrem val zawierającym adres ciągu instrukcji, który powoduje umieszczenie wyniku w val i kończy się skokiem (goto (reg continue)). Rozpoczynając od tego punktu wejścia, skaczemy pod adres wyznaczony przez val, ale wcześniej ustawiamy continue tak, aby po powrocie sterowanie zostało przekazane do print-result, gdzie zostanie wypisana wartość przechowywana w val, a następnie zostanie przekazane do początku pętli evaluatora wczytaj-oblicz-wypisz<sup>50</sup>.

```
external-entry
  (perform (op initialize-stack))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (reg val))
```

Możemy teraz użyć następującej procedury do skompilowania definicji procedury, wykonania skompilowanego kodu i uruchomienia pętli wczytaj-oblicz-wypisz, tak abyśmy mogli przetestować daną procedurę. Ponieważ chcemy, aby skompilowany kod powracał do miejsca określonego przez continue, a wynik znajdował się w rejestrze val, komplilujemy wyrażenie dla rejestrów docelowego val i łącznika return. Chcąc przekształcić kod wynikowy wygenerowany przez kompilator w instrukcje wykonywalne przez maszynę rejestrową evaluatora, używamy procedury assemble symulatora maszyny rejestrowej

---

```
(define (start-eceval)
  (set! the-global-environment (setup-environment))
  (set-register-contents! eceval 'flag false)
  (start eceval))
```

<sup>50</sup> Ponieważ skompilowana procedura jest obiektem, który system może próbować wypisać, zatem modyfikujemy również systemową operację wypisywania user-print (z punktu 4.1.4), tak aby nie próbowała wypisywać składowych skompilowanej procedury:

```
(define (user-print object)
  (cond ((compound-procedure? object)
         (display (list 'compound-procedure
                       (procedure-parameters object)
                       (procedure-body object)
                       '<procedure-env>)))
        ((compiled-procedure? object)
         (display '<compiled-procedure>))
        (else (display object))))
```

(punkt 5.2.2). Następnie inicjujemy rejestr `val`, aby wskazywał na listę instrukcji, ustawiamy rejestr `flag`, aby evaluator skoczył do `external-entry`, i uruchamiamy evaluator.

```
(define (compile-and-go expression)
  (let ((instructions
         (assemble (statements
                     (compile expression 'val 'return)
                     eceval))))
    (set! the-global-environment (setup-environment))
    (set-register-contents! eceval 'val instructions)
    (set-register-contents! eceval 'flag true)
    (start eceval)))
```

Jeśli zainstalowaliśmy monitorowanie stosu, opisane pod koniec punktu 5.4.4, możemy badać wykorzystanie stosu przez skompilowany kod:

```
(compile-and-go
 '(define (factorial n)
   (if (= n 1)
       1
       (* (factorial (- n 1)) n))))
;; Łączna liczba włożeń = 0 maksymalny rozmiar = 0
;; EC-Eval wartość:
ok
;; EC-Eval wprowadź wyrażenie:
(factorial 5)
;; Łączna liczba włożeń = 31 maksymalny rozmiar = 14
;; EC-Eval wartość:
120
```

Porównajmy ten przykład z obliczeniem `(factorial 5)` przy użyciu interpretowanej wersji tej procedury, pokazanym pod koniec punktu 5.4.4. Interpretowana wersja wymagała 144 operacji wkładania na stos, a maksymalny rozmiar stosu był równy 28. Ilustruje to optymalizacje wynikające z przyjętej techniki kompilacji.

### Interpretacja i kompilacja

Dysponując programami przedstawionymi w tym punkcie, możemy teraz eksperymentować, wykonując programy na dwa możliwe sposoby: interpretując je lub kompilując<sup>51</sup>. Interpreter podnosi maszynę do poziomu programów użytkownika; kompilator sprowadza programy użytkownika do poziomu języka

<sup>51</sup> Możemy nawet rozszerzyć kompilator, umożliwiając wywoływanie w skompilowanym kodzie procedur interpretowanych. Zobacz ćwiczenie 5.47.

maszynowego. Możemy traktować język Scheme (lub dowolny język programowania) jak spójną rodzinę abstrakcji opartych na języku maszynowym. Interpretery są dobre do interakcyjnego opracowywania i odpluskowania programów, ponieważ kolejne kroki wykonania programu są wyrażone za pomocą tychże abstrakcji i są dzięki temu bardziej zrozumiałe dla programistów. Skompilowany kod może z kolei być wykonywany szybciej, ponieważ kolejne kroki wykonania programu są wyrażone za pomocą języka maszynowego i kompilator ma swobodę wykonywania optymalizacji, które prowadzą na skróty przez abstrakcje wyższego poziomu<sup>52</sup>.

Możliwość wyboru między interpretacją a komplikacją prowadzi również do różnych technik przenoszenia języków na nowe komputery. Przypuśćmy, że chcemy zaimplementować na nowej maszynie język Lisp. Jedna z możliwości polega na wyjściu od evaluatora z jawnie określonym sterowaniem (z podrozdziału 5.4) i przetłumaczeniu jego instrukcji na instrukcje nowej maszyny. Inne podejście polega na wzięciu kompilatora i przerobieniu jego generatorów kodu tak, aby generowały kod dla nowej maszyny. To drugie podejście pozwala nam na uruchamianie dowolnych programów w Lispie na nowej maszynie, jeżeli je najpierw skompilujemy kompilatorem działającym w oryginalnym systemie lispowym, a następnie połączymy ze skompilowaną wersją biblioteki programów środowiska wykonania<sup>53</sup>. Jeszcze lepiej, możemy skompilować

<sup>52</sup> Niezależnie od techniki wykonywania programów, jeśli nalegamy, aby błędy pojawiające się w trakcie wykonania programu użytkownika były wykrywane i zgłasiane, a nie powodowały awarii systemu czy wygenerowania złych wyników, musimy liczyć się ze znacznym narzutem. Na przykład odwołania poza tablicę mogą być wykrywane poprzez sprawdzanie poprawności odwołania przed jego wykonaniem. Jednak koszt związany ze sprawdzaniem poprawności może wielokrotnie przewyższać koszt samego odwołania do tablicy, w związku z czym programista, decydując, czy takie wykrywanie błędów jest pożądane, powinien rozważyć, czy bardziej mu zależy na szybkości, czy na bezpieczeństwie. Dobry kompilator powinien umożliwiać generowanie kodu z wykrywaniem błędów, powinien unikać nadmiarowych sprawdzień i powinien umożliwiać programiście kontrolę stopnia i rodzaju wykrywanych błędów w skompilowanym kodzie.

Kompilatory popularnych języków, takich jak C czy C++, umieszczają w wykonywanym kodzie niewielkie operacje wykrywające błędy, tak aby wszystko było wykonywane możliwie jak najszybciej. W rezultacie to na programistów spada jawnia realizacja wykrywania błędów. Niestety, ludzie często to zaniedbują, nawet w krytycznych zastosowaniach, gdzie szybkość nie jest najważniejsza. Ich programy wiodą szybki i niebezpieczny żywot. Na przykład słynny „robak”, który sparaliżował Internet w 1988 r., wykorzystywał błąd w systemie operacyjnym UNIX™ polegający na tym, że demon udostępniający informacje na temat użytkowników (ang. *finger daemon*) nie sprawdzał przepełnienia bufora wejściowego. (Zobacz [91]).

<sup>53</sup> Oczywiście zarówno w przypadku wyboru interpretera, jak i kompilatora musimy również zaimplementować na nowej maszynie przydział pamięci, operacje wejścia-wyjścia i wszystkie te najrozmaitsze operacje, co do których założyliśmy, na potrzeby omówienia evaluatora i kompilatora, że są „pierwotne”. Jeden ze sposobów na zminimalizowanie nakładów pracy polega na zapisaniu możliwie jak największej części tych operacji w Lispie, a następnie skompilowaniu ich na nową maszynę. Ostatecznie jednak wszystkie one sprawdzają się do niewielkiego jądra (zawierającego takie operacje, jak odśmiecanie i mechanizm stosowania faktycznych operacji pierwotnych maszyny), które trzeba ręcznie zakodować w języku nowej maszyny.

kompilator nim samym, uruchomić go na nowej maszynie i móc kompilować nim inne programy w Lispie<sup>54</sup>. Możemy też skompilować jeden z interpretatorów z podrozdziału 4.1, uzyskując w wyniku interpreter, który może być uruchamiany na nowej maszynie.

### Ćwiczenie 5.45

Porównując operacje stosowe wykonywane dla tego samego obliczenia przez skompilowany kod i przez evaluator, możemy określić, w jakim stopniu kompilator optymalizuje wykorzystanie stosu zarówno pod względem szybkości (zmniejszając łączną liczbę operacji stosowych), jak i zajętości pamięci (zmniejszając maksymalny rozmiar stosu). Porównanie zoptymalizowanego wykorzystania stosu z wydajnością wyspecjalizowanej maszyny wykonującej to samo obliczenie jest pewnym wskazaniem co do jakości kompilatora.

(a) W ćwiczeniu 5.27 trzeba było określić w zależności od  $n$  liczbę włożeń na stos i maksymalny rozmiar stosu w trakcie obliczania  $n!$  za pomocą podanej tam rekurencyjnej procedury obliczającej silnię. W ćwiczeniu 5.14 należało wykonać takie same pomiary dla wyspecjalizowanej maszyny obliczającej silnię, pokazanej na rys. 5.11. Teraz wykonaj taką samą analizę dla skompilowanej procedury `factorial`.

Wyznacz proporcję liczby włożeń na stos w skompilowanym i interpretowanym programie oraz taką samą proporcję maksymalnych rozmiarów stosu. Ponieważ zarówno liczba operacji, jak i wielkość stosu potrzebnego do obliczenia  $n!$  są liniowe ze względu na  $n$ , zatem w miarę wzrostu  $n$  proporcje te powinny zbiegać do stałych. Jakie to są stałe? Podobnie, wyznacz proporcje wykorzystania stosu w wyspecjalizowanej maszynie i interpretowanym programie.

Porównaj proporcje dla wyspecjalizowanej maszyny i interpretowanego kodu do proporcji dla skompilowanego i interpretowanego kodu. Powinno się okazać, że wyspecjalizowana maszyna jest dużo lepsza niż skompilowany kod, ponieważ skrojony na miarę kod sterownika powinien być dużo lepszy od kodu generowanego przez nasz elementarny kompilator ogólnego przeznaczenia.

(b) Czy potrafisz zaproponować ulepszenia kompilatora, które pomogłyby w generowaniu kodu, który byłby bardziej zbliżony pod względem wydajności do kodu skrójonego na miarę?

### Ćwiczenie 5.46

Przeprowadź analizę podobną do tej z ćwiczenia 5.45, porównując efektywność skompilowanej rozgałęzającej się rekurencyjnie procedury obliczającej liczby Fibonacciego:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

<sup>54</sup> Takie postępowanie prowadzi do zadziwiających testów poprawności kompilatora, takich jak sprawdzenie, czy program skompilowany na nowej maszynie za pomocą skompilowanego kompilatora jest identyczny z tym samym programem, ale skompilowanym pod oryginalnym systemem lispowym. Tropienie źródeł ewentualnych różnic jest niezłą zabawą, ale jest też często frustrujące, ponieważ wyniki są niezwykle czułe na najdrobniejsze szczegóły.

z efektywnością wyspecjalizowanej maszyny obliczającej liczby Fibonacciego, przedstawionej na rys 5.12. (Sposób pomiaru efektywności interpretowanej procedury została przedstawiony w ćwiczeniu 5.29). W przypadku procedury `fib` czas obliczeń nie jest liniowy ze względu na  $n$ , więc proporcje liczb operacji stosowych nie będą zbiegać do niezależnej od  $n$  granicy.

### Ćwiczenie 5.47

W tym punkcie opisaliśmy, jak zmodyfikować evaluator z jawnie określonym sterowaniem, aby interpretowany kod mógł wywoływać skompilowane procedury. Pokaż, jak zmodyfikować kompilator, aby skompilowane procedury mogły wywoływać nie tylko procedury pierwotne i inne skompilowane procedury, lecz także procedury interpretowane. Wymaga to takiego zmodyfikowania `compile-procedure-call`, aby obsługiwany był przypadek (interpretowanych) procedur złożonych. Niemniej jednak upewnij się, że kombinacje `target` i `linkage` są przetwarzane tak jak w `compile-proc-appl`. W celu samego zastosowania procedury w kodzie musi mieć miejsce skok do punktu wejścia `compound-apply` w evaluatorze. Jednak nie możemy bezpośrednio odwoływać się do tej etykiety w kodzie wynikowym (ponieważ assembler wymaga, aby wszystkie etykiety, do których odwołuje się przetwarzany przez niego kod, były w nim zdefiniowane), dlatego też wprowadzimy do maszyny evaluatora rejestr o nazwie `compapp`, w którym będzie przechowywany ten punkt wejścia, i dodamy instrukcje inicjujące go:

```
(assign compapp (label compound-apply))
(branch (label external-entry))      ; skacze, jeśli flag jest ustawiony
read-eval-print-loop
...
```

Aby przetestować swój kod, zdefiniuj najpierw procedurę `f`, która wywołuje procedurę `g`. Skompiluj definicję `f` za pomocą `compile-and-go` i uruchom evaluator. Następnie wprowadź do evaluatora definicję `g` i spróbuj wywołać `f`.

### Ćwiczenie 5.48

Zaimplementowany w niniejszym punkcie interfejs `compile-and-go` jest niewygodny, ponieważ kompilator może być wywoływany tylko raz (w momencie uruchomienia maszyny evaluatora). Rozszerz interfejs kompilator-interpreter, udostępniając operację pierwotną `compile-and-run`, którą można wywoływać w następujący sposób z evaluatora z jawnie określonym sterowaniem:

```
;;; EC-Eval wprowadź wyrażenie:
(compile-and-run
 '(define (factorial n)
   (if (= n 1)
       1
       (* (factorial (- n 1)) n))))
;;; EC-Eval wartość:
ok
```

```
;; EC-Eval wprowadź wyrażenie:  
(factorial 5)  
;; EC-Eval wartość:  
120
```

### Ćwiczenie 5.49

Jako alternatywę dla pętli wczytaj-oblicz-wypisz evaluatora z jawnie określonym sterowaniem zaprojektuj maszynę rejestrową wykonującą pętlę wczytaj-skompiluj-wykonaj-wypisz. Oznacza to, że maszyna powinna wykonywać pętlę, która wczytuje wyrażenie, kompiluje je, asembluje i wykonuje powstały kod, po czym wypisuje wyniki. Jest to łatwe do uruchomienia w naszym symulowanym układzie, ponieważ możemy w nim wywoływać procedury `compile` i `assemble` jako „operacje maszyny rejestrowej”.

### Ćwiczenie 5.50

Skompiluj metacykliczny evaluator z podrozdziału 4.1 za pomocą kompilatora i uruchom go, używając symulatora maszyny rejestrowej. (Aby kompilować więcej niż jedną definicję w danej chwili, możesz umieścić te definicje w wyrażeniu `begin`). Taki interpreter będzie działał bardzo wolno ze względu na wiele poziomów interpretowania, niemniej uruchomienie wszystkich elementów razem to pouczające ćwiczenie.

### Ćwiczenie 5.51

Opracuj elementarną implementację języka Scheme w języku C (lub jakimś innym języku niskiego poziomu), tłumacząc evaluatora z jawnie określonym sterowaniem z podrozdziału 5.4 na język C. W celu uruchomienia tego kodu będziesz musiał również udostępnić odpowiednie procedury zarządzające pamięcią i inne elementy środowiska wykonywania.

### Ćwiczenie 5.52

Jako kontrapunkt do ćwiczenia 5.51 zmodyfikuj kompilator tak, aby kompilując procedury w języku Scheme generował ciągi instrukcji w języku C. Skompiluj metacykliczny evaluator z podrozdziału 4.1, uzyskując interpreter języka Scheme napisany w języku C.

# Literatura

- [1] Abelson, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman, and Jack Wisdom. 1992. The Supercomputer Toolkit: A general framework for special-purpose computing. *International Journal of High-Speed Electronics* 3(3):337–361.
- [2] Allen, John. 1978. *Anatomy of Lisp*. New York: McGraw-Hill.
- [3] ANSI X3.226-1994. *American National Standard for Information Systems—Programming Language—Common Lisp*.
- [4] Appel, Andrew W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25(4):275–279.
- [5] Backus, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8):613–641.
- [6] Baker, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4):280–293.
- [7] Batali, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman, and Daniel Weise. 1982. The Scheme-81 architecture—System and chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, edited by Paul Penfield, Jr. Dedham, MA: Artech House.
- [8] Borning, Alan. 1977. ThingLab—An object-oriented system for building simulations using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*.
- [9] Borodin, Alan, and Ian Munro. 1975. *The Computational Complexity of Algebraic and Numeric Problems*. New York: American Elsevier.
- [10] Chaitin, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5):47–52.
- [11] Church, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton, N.J.: Princeton University Press.
- [12] Clark, Keith L. 1978. Negation as failure. In *Logic and Data Bases*. New York: Plenum Press, pp. 293–322.

- [13] Clinger, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 226–234.
- [14] Clinger, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pp. 155–162.
- [15] Colmerauer A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy.
- [16] Cormen, Thomas, Charles Leiserson, Ronald Rivest. *Wprowadzenie do algorytmów*. Wyd. 4. WNT, Warszawa 2001.
- [17] Darlington, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications*. New York: Cambridge University Press.
- [18] Dijkstra, Edsger W. 1968a. The structure of the “THE” multiprogramming system. *Communications of the ACM* 11(5):341–346.
- [19] Dijkstra, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43–112.
- [20] Dinesman, Howard P. 1968. *Superior Mathematical Puzzles*. New York: Simon and Schuster.
- [21] deKleer, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pp. 116–125.
- [22] Doyle, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12:231–272.
- [23] Feigenbaum, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems*, vol. 9, pp. 105–117.
- [24] Feeley, Marc. 1986. Deux approches à l’implantation du language Scheme. Masters thesis, Université de Montréal.
- [25] Feeley, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1):47–66.
- [26] Feller, William. 1957. *An Introduction to Probability Theory and Its Applications*, volume 1. New York: John Wiley & Sons.
- [27] Fenichel, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11):611–612.
- [28] Floyd, Robert. 1967. Nondeterministic algorithms. *JACM*, 14(4):636–644.
- [29] Forbus, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press.

- 
- [30] Friedman, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium*, edited by S. Michaelson and R. Milner, pp. 257–284.
  - [31] Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages*. Cambridge, MA: MIT Press/McGraw-Hill.
  - [32] Gabriel, Richard P. 1988. The Why of *Y*. *Lisp Pointers* 2(2):15–25.
  - [33] Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley.
  - [34] Gordon, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag.
  - [35] Gray, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models*. San Mateo, CA: Morgan-Kaufman.
  - [36] Green, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 219–240.
  - [37] Green, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the ACM National Conference*, pp. 169–181.
  - [38] Griss, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah.
  - [39] Guttag, John V. 1977. Abstract data types and the development of data structures. *Communications of the ACM* 20(6):397–404.
  - [40] Hamming, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, N.J.: Prentice-Hall.
  - [41] Hanson, Christopher P. 1990. Efficient stack allocation for tail-recursive languages. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pp. 106–118.
  - [42] Hanson, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers*, 4(3).
  - [43] Hardy, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2).
  - [44] Hardy, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers*. 4th edition. New York: Oxford University Press.
  - [45] Havender, J. 1968. Avoiding deadlocks in multi-tasking systems. *IBM Systems Journal* 7(2):74–84.
  - [46] Hearn, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University.

- 
- [47] Henderson, Peter. 1980. *Functional Programming: Application and Implementation*. Englewood Cliffs, N.J.: Prentice-Hall.
  - [48] Henderson, Peter. 1982. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 179–187.
  - [49] Hewitt, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 295–301.
  - [50] Hewitt, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3):323–364.
  - [51] Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1).
  - [52] Hodges, Andrew. *ENIGMA. Życie i śmierć Alana Turinga*. Prószyński i S-ka, Warszawa 2002.
  - [53] Hofstadter, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.
  - [54] Hughes, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17–42.
  - [55] IEEE Std 1178–1990. 1990. *IEEE Standard for the Scheme Programming Language*.
  - [56] Ingerman, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Unpublished manuscript. (Also, private communication from Wallace Feurzeig ).
  - [57] Kaldewaij, Anne. 1990. *Programming: The Derivation of Algorithms*. New York: Prentice-Hall.
  - [58] Knuth, Donald E. *Sztuka programowania. Tom 1: Algorytmy podstawowe*. WNT, Warszawa 2002.
  - [59] Knuth, Donald E. *Sztuka programowania. Tom 2: Algorytmy seminumeryczne*. WNT, Warszawa 2002.
  - [60] Kohlbecker, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University.
  - [61] Konopasek, Milos, and Sundaresan Jayaraman. 1984. *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Berkeley, CA: Osborne/McGraw-Hill.
  - [62] Kowalski, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh.

- [63] Kowalski, Robert. 1979. *Logic for Problem Solving*. New York: North-Holland.
- [64] Lamport, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558–565.
- [65] Lampson, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek. 1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto.
- [66] Landin, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101.
- [67] Lieberman, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6):419–429.
- [68] Liskov, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* 1(1):7–19.
- [69] McAllester, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory.
- [70] McAllester, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory.
- [71] McCarthy, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4):184–195.
- [72] McCarthy, John. 1967. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland.
- [73] McCarthy, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages*.
- [74] McCarthy, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual*. 2nd edition. Cambridge, MA: MIT Press.
- [75] McDermott, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory.
- [76] Miller, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3):300–317.
- [77] Miller, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory.
- [78] Moon, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science.
- [79] Moon, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory.

- 
- [80] Morris, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages*.
  - [81] Phillips, Hubert. 1934. *The Sphinx Problem Book*. London: Faber and Faber.
  - [82] Pitman, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science.
  - [83] Rabin, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12:128–138.
  - [84] Raymond, Eric. 1993. *The New Hacker's Dictionary*. 2nd edition. Cambridge, MA: MIT Press.
  - [85] Raynal, Michel. 1986. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press.
  - [86] Rees, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114–122.
  - [87] Rees, Jonathan, and William Clinger (eds). 1991. The revised<sup>4</sup> report on the algorithmic language Scheme. *Lisp Pointers*, 4(3).
  - [88] Rivest, Ronald, Adi Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LCS/TM82, MIT Laboratory for Computer Science.
  - [89] Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1):23.
  - [90] Robinson, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1:107–124.
  - [91] Spafford, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the ACM* 32(6):678–688.
  - [92] Steele, Guy Lewis, Jr. 1977. Debunking the “expensive procedure call” myth. In *Proceedings of the National Conference of the ACM*, pp. 153–162.
  - [93] Steele, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 98–107.
  - [94] Steele, Guy Lewis, Jr. 1990. *Common Lisp: The Language*. 2nd edition. Digital Press.
  - [95] Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory.
  - [96] Steele, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker's Dictionary*. New York: Harper & Row.

- [97] Stoy, Joseph E. 1977. *Denotational Semantics*. Cambridge, MA: MIT Press.
- [98] Sussman, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided circuit analysis. *IEEE Transactions on Circuits and Systems* CAS-22(11):857–865.
- [99] Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierarchical descriptions. *AI Journal* 14:1–39.
- [100] Sussman, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257:256–262.
- [101] Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory.
- [102] Sutherland, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory.
- [103] Teitelman, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center.
- [104] Thatcher, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 119–132.
- [105] Turner, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics*, Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334–348.
- [106] Wand, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):164–180.
- [107] Waters, Richard C. 1979. A method for analyzing loop programs. *IEEE Transactions on Software Engineering* 5(3):237–247.
- [108] Winograd, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory.
- [109] Winston, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley.
- [110] Zabih, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. *AAAI-87*, pp. 59–64.
- [111] Zippel, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT.
- [112] Zippel, Richard. 1993. *Effective Polynomial Computation*. Boston, MA: Kluwer Academic Publishers.

# Wykaz ćwiczeń

<b>1.1</b>	19	<b>1.11</b>	40	<b>1.21</b>	52	<b>1.31</b>	59	<b>1.41</b>	76
<b>1.2</b>	20	<b>1.12</b>	41	<b>1.22</b>	52	<b>1.32</b>	60	<b>1.42</b>	76
<b>1.3</b>	20	<b>1.13</b>	41	<b>1.23</b>	53	<b>1.33</b>	60	<b>1.43</b>	76
<b>1.4</b>	20	<b>1.14</b>	42	<b>1.24</b>	53	<b>1.34</b>	65	<b>1.44</b>	77
<b>1.5</b>	20	<b>1.15</b>	43	<b>1.25</b>	53	<b>1.35</b>	69	<b>1.45</b>	77
<b>1.6</b>	24	<b>1.16</b>	45	<b>1.26</b>	54	<b>1.36</b>	69	<b>1.46</b>	77
<b>1.7</b>	25	<b>1.17</b>	45	<b>1.27</b>	54	<b>1.37</b>	70		
<b>1.8</b>	25	<b>1.18</b>	46	<b>1.28</b>	54	<b>1.38</b>	70		
<b>1.9</b>	35	<b>1.19</b>	46	<b>1.29</b>	59	<b>1.39</b>	71		
<b>1.10</b>	35	<b>1.20</b>	48	<b>1.30</b>	59	<b>1.40</b>	76		
<b>2.1</b>	87	<b>2.21</b>	106	<b>2.41</b>	124	<b>2.61</b>	154	<b>2.81</b>	197
<b>2.2</b>	89	<b>2.22</b>	106	<b>2.42</b>	124	<b>2.62</b>	154	<b>2.82</b>	198
<b>2.3</b>	90	<b>2.23</b>	107	<b>2.43</b>	125	<b>2.63</b>	157	<b>2.83</b>	198
<b>2.4</b>	92	<b>2.24</b>	109	<b>2.44</b>	132	<b>2.64</b>	157	<b>2.84</b>	198
<b>2.5</b>	92	<b>2.25</b>	109	<b>2.45</b>	133	<b>2.65</b>	158	<b>2.85</b>	198
<b>2.6</b>	92	<b>2.26</b>	109	<b>2.46</b>	135	<b>2.66</b>	159	<b>2.86</b>	199
<b>2.7</b>	94	<b>2.27</b>	110	<b>2.47</b>	135	<b>2.67</b>	166	<b>2.87</b>	206
<b>2.8</b>	94	<b>2.28</b>	110	<b>2.48</b>	136	<b>2.68</b>	166	<b>2.88</b>	206
<b>2.9</b>	94	<b>2.29</b>	110	<b>2.49</b>	136	<b>2.69</b>	166	<b>2.89</b>	206
<b>2.10</b>	95	<b>2.30</b>	112	<b>2.50</b>	139	<b>2.70</b>	167	<b>2.90</b>	206
<b>2.11</b>	95	<b>2.31</b>	112	<b>2.51</b>	139	<b>2.71</b>	167	<b>2.91</b>	206
<b>2.12</b>	95	<b>2.32</b>	112	<b>2.52</b>	140	<b>2.72</b>	167	<b>2.92</b>	208
<b>2.13</b>	96	<b>2.33</b>	118	<b>2.53</b>	143	<b>2.73</b>	182	<b>2.93</b>	209
<b>2.14</b>	96	<b>2.34</b>	118	<b>2.54</b>	143	<b>2.74</b>	183	<b>2.94</b>	210
<b>2.15</b>	96	<b>2.35</b>	119	<b>2.55</b>	144	<b>2.75</b>	185	<b>2.95</b>	210
<b>2.16</b>	97	<b>2.36</b>	119	<b>2.56</b>	149	<b>2.76</b>	185	<b>2.96</b>	211
<b>2.17</b>	102	<b>2.37</b>	119	<b>2.57</b>	149	<b>2.77</b>	190	<b>2.97</b>	212
<b>2.18</b>	102	<b>2.38</b>	120	<b>2.58</b>	149	<b>2.78</b>	190		
<b>2.19</b>	102	<b>2.39</b>	121	<b>2.59</b>	152	<b>2.79</b>	191		
<b>2.20</b>	103	<b>2.40</b>	124	<b>2.60</b>	152	<b>2.80</b>	191		
<b>3.1</b>	220	<b>3.5</b>	224	<b>3.9</b>	238	<b>3.13</b>	251	<b>3.17</b>	254
<b>3.2</b>	220	<b>3.6</b>	225	<b>3.10</b>	243	<b>3.14</b>	251	<b>3.18</b>	254
<b>3.3</b>	221	<b>3.7</b>	231	<b>3.11</b>	245	<b>3.15</b>	253	<b>3.19</b>	254
<b>3.4</b>	221	<b>3.8</b>	231	<b>3.12</b>	250	<b>3.16</b>	253	<b>3.20</b>	255

<b>3.21</b>	259	<b>3.34</b>	288	<b>3.47</b>	307	<b>3.60</b>	326	<b>3.73</b>	337
<b>3.22</b>	260	<b>3.35</b>	288	<b>3.48</b>	308	<b>3.61</b>	327	<b>3.74</b>	338
<b>3.23</b>	260	<b>3.36</b>	289	<b>3.49</b>	308	<b>3.62</b>	327	<b>3.75</b>	339
<b>3.24</b>	265	<b>3.37</b>	289	<b>3.50</b>	318	<b>3.63</b>	331	<b>3.76</b>	339
<b>3.25</b>	266	<b>3.38</b>	296	<b>3.51</b>	318	<b>3.64</b>	331	<b>3.77</b>	342
<b>3.26</b>	266	<b>3.39</b>	299	<b>3.52</b>	319	<b>3.65</b>	332	<b>3.78</b>	342
<b>3.27</b>	266	<b>3.40</b>	299	<b>3.53</b>	324	<b>3.66</b>	334	<b>3.79</b>	342
<b>3.28</b>	271	<b>3.41</b>	300	<b>3.54</b>	324	<b>3.67</b>	335	<b>3.80</b>	342
<b>3.29</b>	272	<b>3.42</b>	300	<b>3.55</b>	324	<b>3.68</b>	335	<b>3.81</b>	347
<b>3.30</b>	272	<b>3.43</b>	303	<b>3.56</b>	324	<b>3.69</b>	335	<b>3.82</b>	347
<b>3.31</b>	276	<b>3.44</b>	303	<b>3.57</b>	325	<b>3.70</b>	335		
<b>3.32</b>	279	<b>3.45</b>	303	<b>3.58</b>	325	<b>3.71</b>	336		
<b>3.33</b>	288	<b>3.46</b>	307	<b>3.59</b>	326	<b>3.72</b>	336		
<b>4.1</b>	360	<b>4.17</b>	381	<b>4.33</b>	402	<b>4.49</b>	417	<b>4.65</b>	458
<b>4.2</b>	365	<b>4.18</b>	382	<b>4.34</b>	403	<b>4.50</b>	427	<b>4.66</b>	458
<b>4.3</b>	366	<b>4.19</b>	382	<b>4.35</b>	408	<b>4.51</b>	428	<b>4.67</b>	459
<b>4.4</b>	366	<b>4.20</b>	383	<b>4.36</b>	409	<b>4.52</b>	428	<b>4.68</b>	459
<b>4.5</b>	366	<b>4.21</b>	384	<b>4.37</b>	409	<b>4.53</b>	429	<b>4.69</b>	459
<b>4.6</b>	367	<b>4.22</b>	389	<b>4.38</b>	411	<b>4.54</b>	429	<b>4.70</b>	474
<b>4.7</b>	367	<b>4.23</b>	389	<b>4.39</b>	411	<b>4.55</b>	437	<b>4.71</b>	479
<b>4.8</b>	367	<b>4.24</b>	389	<b>4.40</b>	411	<b>4.56</b>	439	<b>4.72</b>	479
<b>4.9</b>	368	<b>4.25</b>	392	<b>4.41</b>	411	<b>4.57</b>	441	<b>4.73</b>	479
<b>4.10</b>	368	<b>4.26</b>	392	<b>4.42</b>	411	<b>4.58</b>	441	<b>4.74</b>	479
<b>4.11</b>	372	<b>4.27</b>	397	<b>4.43</b>	412	<b>4.59</b>	441	<b>4.75</b>	480
<b>4.12</b>	372	<b>4.28</b>	398	<b>4.44</b>	412	<b>4.60</b>	442	<b>4.76</b>	481
<b>4.13</b>	372	<b>4.29</b>	398	<b>4.45</b>	416	<b>4.61</b>	444	<b>4.77</b>	481
<b>4.14</b>	376	<b>4.30</b>	398	<b>4.46</b>	417	<b>4.62</b>	444	<b>4.78</b>	481
<b>4.15</b>	378	<b>4.31</b>	399	<b>4.47</b>	417	<b>4.63</b>	444	<b>4.79</b>	481
<b>4.16</b>	381	<b>4.32</b>	402	<b>4.48</b>	417	<b>4.64</b>	458		
<b>5.1</b>	487	<b>5.12</b>	521	<b>5.23</b>	550	<b>5.34</b>	585	<b>5.45</b>	599
<b>5.2</b>	490	<b>5.13</b>	521	<b>5.24</b>	550	<b>5.35</b>	585	<b>5.46</b>	599
<b>5.3</b>	494	<b>5.14</b>	523	<b>5.25</b>	551	<b>5.36</b>	585	<b>5.47</b>	600
<b>5.4</b>	503	<b>5.15</b>	523	<b>5.26</b>	554	<b>5.37</b>	585	<b>5.48</b>	600
<b>5.5</b>	503	<b>5.16</b>	523	<b>5.27</b>	555	<b>5.38</b>	589	<b>5.49</b>	601
<b>5.6</b>	504	<b>5.17</b>	523	<b>5.28</b>	555	<b>5.39</b>	592	<b>5.50</b>	601
<b>5.7</b>	507	<b>5.18</b>	524	<b>5.29</b>	555	<b>5.40</b>	592	<b>5.51</b>	601
<b>5.8</b>	514	<b>5.19</b>	524	<b>5.30</b>	556	<b>5.41</b>	593	<b>5.52</b>	601
<b>5.9</b>	520	<b>5.20</b>	530	<b>5.31</b>	564	<b>5.42</b>	593		
<b>5.10</b>	520	<b>5.21</b>	530	<b>5.32</b>	564	<b>5.43</b>	593		
<b>5.11</b>	520	<b>5.22</b>	531	<b>5.33</b>	585	<b>5.44</b>	594		

# Skorowidz

Wszelkie nieścisłości w niniejszym skorowidzu można wyjaśnić tym, że został on przygotowany za pomocą komputera.

Donald E. Knuth, *Sztuka programowania*  
*Tom I: Algorytmy podstawowe*\*

- \* (pierwotna procedura mnożenia) 5
- + (pierwotna procedura dodawania) 5
- (pierwotna procedura odejmowania) 5
  - zmiana znaku liczby 17
- / (pierwotna procedura dzielenia) 6
- = (pierwotna równość liczb) 17
- =number? 148
- =zero? predykat ogólny 191
  - dla wielomianów 206
- < (pierwotna relacja porządku) 17
- > (pierwotna relacja porządku) 17
- >= 19
- ; zob. średnik
- ! w nazwach 216
- ? w nazwach predykatów 23
- " (cudzysłów) 142
- ' (apostrof) 142
  - makrodefinicja read 374, 477
- ' (wsteczny apostrof) 565
- , (przecinek, używany razem ze wstecznym apostrofem) 565
- #f 17
- #t 17
- $\lambda$  zob. rachunek lambda
- $\pi$  zob. pi
- $\theta$  zob. theta
- Abelson Harold 3
- abs **16, 18**
- abstrakcja 4
  - danych 80, 83, 167, 171, 360
  - kolejki 256
  - metajęzykowa 352
- abstrakcja 4
  - proceduralna 26
  - w projektach maszyn rejestrowych 491–494
- wspólne wzory a - 56
- wyszukiwania w programach niedeterministycznych 409
- accelerated-sequence **330**
- accumulate **60, 115**
  - to samo co fold-right 120
- accumulate-n 119
- Áchárya Bháscara 41
- actual-value **394**
- Ada 444
  - procedury rekurencyjne 34
- add (procedura ogólna) **186**
  - dodawanie współczynników wielomianów 203, 204
- add-action! 270, **273**
- add-binding-to-frame! **370**
- add-complex **171**
- add-complex-to-schemenum **191**
- add-interval **93**
- add-lists **401**
- add-poly **201**
- add-rat 84
- add-rule-or-assertion! **473**
- add-streams **322**
- add-terms **203**
- add-to-agenda! 274, **278**
- add-vect 135
- addend **147**
- adder (więzy pierwotne) **283**
- addytywność 82, 168, 177–183, 188

\* Cytat pochodzi z drugiego amerykańskiego wydania tego dzieła z 1973 r. (przyp. red.).

- Adelman Leonard 52  
**adjoin-arg** 542  
**adjoin-set** 150  
 dla zbiorów ważonych 165  
 reprezentacja zbiorów za pomocą drzew binarnych 156  
 reprezentacja zbiorów za pomocą list nieuporządkowanych 151  
 reprezentacja zbiorów za pomocą list uporządkowanych 154  
**adjoin-term** 202, 205  
**adres** 525  
 arytmetyka 525  
 bazowy wektora 526  
 przekierowania 533  
 składowy 591  
 adresowanie składowe 590–592  
**advance-pc** 516  
**after-delay** 271, 274  
**A'�-mose** 46  
 akcelerator ciągu 329  
 akcje, w maszynie rejestrowej 490–491  
 akumulacja drzewiasta 9  
 akumulator 114, 220  
 algebra symboliczna 199–212  
**Algol**  
 przekazywanie argumentów przez nazwę 318, 393  
 słaby punkt w operowaniu obiektami złożonym 290  
 struktura blokowa 30  
*thunk* 392  
**algorytm**  
 Euklidesa 47–48, 484  
 dla wielomianów 209  
 złożoność 48  
 mnożenia rosyjskich chłopów 46  
 optymalny 119  
 probabilistyczny 51–52, 212, 320  
 RSA 52  
 unifikacji  
 wynalezienie 430  
 aliasowanie 229  
**all-regs** (w kompilatorze) 577  
**Allen** John 532  
 alternatywa wyrażenia warunkowego  
 if 18  
**always-true** 464  
 amb (forma specjalna) 404, 405  
**ambeval** 420  
**an-element-of** 405  
**an-integer-starting-from** 406  
 analiza  
 numeryczna 5  
 przypadków a programowanie sterowane danymi 357  
 przypadków, ogólnie 16  
 składowa  
 języka naturalnego 412–417  
 oddzielona od wykonywania 385–389, 511, 516  
 programów niedeterministycznych 420  
**analyze** metacykliczne 386  
**analyze-**...  
 metacykliczne 386–389  
 niedeterministyczne 421–424  
**analyze-amb** 426  
**and** (forma specjalna) 18  
 bez podwyrażeń 366  
 dlaczego forma specjalna 19  
 obliczanie wartości 18, 366  
**and** (w języku zapytań) 437  
 obliczanie 447, 462, 481  
**angle**  
 dla danych ze znacznikami typu 176  
 programowanie sterowane danymi 181  
 reprezentacja biegunowa liczb zespolonych 173  
 reprezentacja prostokątna liczb zespolonych 172  
**angle-polar** 175  
**angle-rectangular** 175  
**announce-output** 375  
**APL** 118  
 apostrof wsteczny (odwrócony) 565  
**append** 102, 250  
 implementacja w postaci maszyny rejestrowej 531  
 porównanie reguły („co”)  
 z procedurą („jak”) 431–432  
 porównanie z append 250  
 z dowolną liczbą argumentów 580  
 za pomocą kumulacji 118

- append! **250**  
implementacja w postaci maszyny rejestrowej **531**  
**append-instruction-sequences** **562, 579**  
**append-to-form** (reguła) **443**  
**application?** **364**  
**apply** (leniwe) **394**  
**apply** (metacykliczne) **358**  
porównanie z pierwotnym **apply** **378**  
**apply** (procedura pierwotna) **181**  
**apply-dispatch** **544**  
zmodyfikowane pod kątem skompilowanego kodu **595**  
**apply-generic** **181**  
dla wieży typów **195**  
z przekazywaniem argumentów **184**  
z rzutowaniem typów **193, 197, 198**  
dla wielu argumentów **198**  
w góre **198**  
**apply-primitive-procedure** **358, 369, 374**  
**apply-rules** **467**  
**arbiter** **307**  
architektura znacznikowa **527**  
**arcus tangens** **172**  
**argl** (rejestr) **539**  
**argument(y)** **6**  
dowolna liczba **6, 103**  
kombinacji **6**  
odroczone **341**  
**articles** **412**  
**Arystoteles** *De caelo* (komentarz Buridana) **307**  
**arytmetyka**  
adresów **525**  
liczb wymiernych **83–87**  
ogólny system arytmetyczny **187**  
potrzeba danych złożonych **80**  
liczb zespolonych **169**  
składowa ogólnego systemu arytmetycznego **188**  
struktura systemu **177**  
ogólna **185**  
procedury pierwotne **5**  
przedziałów **93–97**  
szeregow potęgowych **326, 327**
- arytmetyka wielomianów **200–212**  
algorytm Euklidesa **209**  
dodawanie **201–204**  
dzielenie **206**  
funkcje wymierne **208–212**  
mnożenie **201–204**  
największy wspólny dzielnik **209–212**  
odejmowanie **206**  
probabilistyczny algorytm obliczania NWD **212**  
w ogólnym systemie arytmetycznym **201**  
*Arytmetyka* Diofantosa **50**  
**ASCII** (kod) **159**  
**asembler** **507, 511–514**  
**asercja** **433**  
**assemble** **512, 513**  
**assert!** (w interpreterze zapytań) **453**  
**assign** (instrukcja maszyny rejestrowej) **489**  
symulacja **515**  
zapamiętywanie etykiety w rejestrze **497**  
**assign-reg-name** **516**  
**assign-value-exp** **516**  
**assignment-value** **361**  
**assignment-variable** **361**  
**assignment?** **361**  
**assoc** **262**  
**atan** (procedura pierwotna) **172**  
**atrapa** **261, 277**  
**attach-tag** **174**  
z użyciem typów danych języka Scheme **190**  
**augend** **147**  
automagicznie **407**  
**average** **23**  
**average-damp** **71**  
**averager** (wieży) **288**
- B-drzewa** **156**  
**Backus John** **349**  
**Baker Henry G. Jr.** **532**  
**bariera synchronizacyjna** **308**  
**bariery abstrakcji** **81, 87–90, 167**  
ogólny system arytmetyczny **185**

- bariery abstrakcji  
 w systemie liczb zespolonych 169  
 Barth John 351  
 Basic  
 ograniczenia dotyczące danych  
 złożonych 98  
 słaby punkt w operowaniu  
 obiektami złożonymi 290  
 baza danych  
 indeksowanie 446, 471  
 personelu Nienasyconego  
 Przedsiębiorstwa SA 183  
 pracowników firmy Mikrusoft  
 433–435  
 programowanie sterowane danymi  
 i - 183  
 programowanie w logice i - 433  
 zbiór rekordów 158  
**begin** (forma specjalna) **216**  
 występujące niejawnie  
 w następnikach klauzul **cond**  
 i w ciałach procedur 217  
**begin-actions** **363**  
**begin?** **363**  
**below** 127, 139  
**beside** 127, **138**  
**bignum** (typ danych) 528  
**bisekcja** 65–67  
**błąd**  
 zaokrąglenia 5, 170  
 zasłanianie zmiennych wolnych 28  
 związany z efektami ubocznymi 229  
 związany z kolejnością przypisań  
 230  
 Bolt Beranek and Newman Inc. 3  
 Borning Alan 280  
 Borodin Alan 119  
 bramka  
 AND 267  
 and-gate **271**  
 OR 268  
 or-gate 271, 272  
 w układzie cyfrowym 267  
**branch** (instrukcja maszyny  
 rejestrowej) 488  
 symulacja 517  
**branch-dest** **517**  
 Buridan Jean 307
- C**  
 interpreter języka Scheme napisany  
 w - 601  
 komplikacja języka Scheme do - 601  
 obsługa błędów 556, 598  
 ograniczenia dotyczące danych  
 złożonych 98  
 procedury rekurencyjne 34  
**ca...r** 100  
**cadr** 100  
**call-each** **273**  
 całka  
 oznaczona 58–59  
 przybliżenie metodą Monte Carlo  
 224, 347  
 szeregu potęgowego 326  
 całkowanie  
 metoda Monte Carlo 224  
 sformułowanie strumieniowe 347  
**car** (procedura pierwotna) **85**  
 aksjomat opisujący 91  
 implementacja proceduralna **91, 92, 254, 255, 401**  
 implementacja wektorowa 528  
 jako operacja na listach 100  
 pochodzenie nazwy 85  
**cd...r** 100  
**cdr** (procedura pierwotna) **85**  
 aksjomat opisujący 91  
 implementacja proceduralna **91, 92, 254, 255, 401**  
 implementacja wektorowa 528  
 jako operacja na listach 100  
 pochodzenie nazwy 85  
**cdr** wzduż listy 100  
**celsius-fahrenheit-converter**  
**281**  
 w stylu zorientowanym na  
 wyrażenia **289**  
**center** **95**  
 Cesàro Ernesto 222  
**cesaro-stream** **346**  
**cesaro-test** **223**  
 Chaitin Gregory 222  
 Chandah-sutra 45  
 Chapman David 408  
 Charniak Eugene 407  
 Chu Shih-chieh 41  
 Church Alonzo 93

- ciąg instrukcji 562–564  
ciagi 98  
jako konwencjonalne interfejsy 112–125  
jako źródło modularności 117  
operacje na - 114–121  
reprezentacja za pomocą par 98  
Clark Keith L. 458  
Clinger William 393  
coeff 202, **206**  
Colmerauer Alain 430  
Common Lisp 3  
traktowanie nil 100  
compile **560**  
compile-and-go 594, **597**  
compile-and-run 600  
compile-application **572**  
compile-assignment **566**  
compile-definition **566**  
compile-if **568**  
compile-lambda **570**  
compile-linkage **565**  
compile-proc-appl **577**  
compile-procedure-call **574**  
compile-quoted **566**  
compile-self-evaluating **565**  
compile-sequence **569**  
compile-variable **566**  
compiled-apply **595**  
compiled-procedure-entry **570**  
compiled-procedure-env **570**  
compiled-procedure? **570**  
complex (pakiet) 188  
complex->complex **197**  
compound-apply **545**  
compound-procedure? **369**  
cond (forma specjalna) **16**  
dodatkowa składnia klauzul **366**  
klauzula 17  
niejawną formą begin  
w następcach klauzul 217  
obliczanie wartości 17  
porównanie z if 18  
cond->if **365**  
cond-actions **365**  
cond-clauses **365**  
cond-else-clause? **365**  
cond-predicate **365**  
cond? **364**
- conjoin **463**  
connect 283, **288**  
Conniver 407  
cons (procedura pierwotna) **84**  
aksjomat opisujący 91  
implementacja  
proceduralna **91, 92, 249, 254, 255, 401**  
wektorowa 529  
za pomocą modyfikatorów 248  
jako operacja na listach 100  
własność domknięcia 97  
znaczenie nazwy **85**  
konstruowanie listy 102  
cons-stream (forma specjalna) **312, 314**  
dlaczego forma specjalna 314  
leniwe obliczanie i - 400  
const (w maszynie rejestrowej) 489  
składnia 505  
symulacja 519  
constant (więzy pierwotne) **286**  
constant-exp **519**  
constant-exp-value **519**  
construct-arglist **573**  
contents **174**  
z użyciem typów danych języka  
Scheme 190  
continue (rejestr) 496  
rekursja i - 500  
w evaluatorze z jawnie określonym  
sterowaniem 539  
Cormen Thomas H. 156  
corner-split **131**  
cos (procedura pierwotna) **68**  
cosinus  
granica szeregu potęgowego 326  
punkt stały 68  
count-change **39**  
count-leaves **107, 109**  
jako maszyna rejestrowa 530  
za pomocą kumulacji 119  
count-pairs **254**  
Cressey David 533  
cube **43, 55, 73**  
cube-root **72**  
current-time **274, 277**  
cykl w liście 251  
wykrywanie 254

- cytowanie 141–144
  - list 142
  - napisów 142
  - obiektów danych w Lispie 141
  - symboli 142
  - w językach naturalnych 141
- czarna skrzynka 26
- czas
  - bieżący, symulacji 277
  - komunikacja i - 309
  - programowanie funkcyjne i - 347–350
  - przypisanie a - 290
  - sens 292
  - w obliczeniach
    - niedeterministycznych 404, 406
  - w systemach współbieżnych 292–296
- Czebyszew Pafnutyj L. 324
- czynnik zachowania współczynników całkowitych 211
- dane 1, 4
  - abstrakcyjne 83
  - hierarchiczne 107–111
  - jako program 376–378
  - konkretna reprezentacja 83
  - liczbowe 5
  - modele abstrakcyjne 90
  - reprezentacja proceduralna 90–93
  - specyfikacje algebraiczne 91
  - symboliczne 140
  - współdzielone 251–254, 293
  - ze znacznikami 173–177, 526
  - złożone 79–81
  - znaczenie 90–93
- decode **164**
- deep-reverse 110
- define (forma specjalna) 7, 10
  - dla procedur **12**, 61
  - lukier syntaktyczny 362
  - notacja kropki i ogona 103
  - porównanie z lambda 61
  - w modelu środowiskowym 236
  - wartość 7
- define-variable! 369, **371**
- definicja lokalna 29–30
  - położenie 30
  - porównanie z let 64
- definicja lokalna
  - wolne zmienne w - 29
- definicja wewnętrzna
  - ograniczenia 380
  - w modelu środowiskowym 243–246
  - w ewaluatorze niedeterministycznym 423
  - wyłuskiwanie 380
    - w kompilatorze 592, 593
  - zakres nazwy 379–381
- definition-value **362**
- definition-variable **362**
- definition? **362**
- deKleer Johan 407
- dekompozycja programu na części 25
- delay (forma specjalna) **314**
  - dłaczego forma specjalna 314
  - implementacja za pomocą lambda-abstrakcji 317
  - jawne użycie 341
  - leniwe obliczanie i - 400
  - ze spamiętywaniem 317, 325
- delay-it **396**
- delete-queue! 256, **259**
- denom 83, 86
  - aksjomat opisujący 90
  - skracanie ułamków **89**
- deposit (operacja na koncie bankowym) **219**
  - z zewnętrznym szeregowaniem **302**
- deriv
  - różniczkowanie numeryczne **73**
  - różniczkowanie symboliczne **146**
    - sterowane danymi **182**
- deskryptor łącznika 561
- diagram
  - przebiegu zdarzeń 293
  - przepływu sygnałów 113, 338
- Dijkstra Edsger Wybe 304
- Dinesman Howard P. 409
- Diofantos 50
- disjoin **463**
- display (procedura pierwotna) 52, **86**
  - display-line **313**
  - display-stream **313**
  - distinct? **410**
- div (procedura ogólna) **186**
- div-complex **172**

- cytowanie 141–144  
 list 142  
 napisów 142  
 obiektów danych w Lispie 141  
 symbole 142  
 w językach naturalnych 141  
 czarna skrzynka 26  
 czas  
     bieżący, symulacji 277  
     komunikacja i - 309  
     programowanie funkcyjne i - 347–350  
     przypisanie a - 290  
     sens 292  
     w obliczeniach  
         niedeterministycznych 404, 406  
     w systemach współbieżnych 292–296  
 Czebyszew Pafnutyj L. 324  
 czynnik zachowania współczynników całkowitych 211  
  
 dane 1, 4  
     abstrakcyjne 83  
     hierarchiczne 107–111  
     jako program 376–378  
     konkretna reprezentacja 83  
     liczbowe 5  
     modele abstrakcyjne 90  
     reprezentacja proceduralna 90–93  
     specyfikacje algebraiczne 91  
     symboliczne 140  
     współdzielone 251–254, 293  
     ze znacznikami 173–177, 526  
     złożone 79–81  
     znaczenie 90–93  
 decode **164**  
 deep-reverse 110  
 define (forma specjalna) 7, 10  
     dla procedur **12**, 61  
     lukier syntaktyczny 362  
     notacja kropki i ogona 103  
     porównanie z lambda 61  
     w modelu środowiskowym 236  
     wartość 7  
 define-variable! 369, **371**  
 definicja lokalna 29–30  
     położenie 30  
     porównanie z let 64  
  
 definicja lokalna  
     wolne zmienne w - 29  
 definicja wewnętrzna  
     ograniczenia 380  
     w modelu środowiskowym 243–246  
     w ewaluatorze niedeterministycznym 423  
     wyłuskiwanie 380  
         w kompilatorze 592, 593  
     zakres nazwy 379–381  
 definition-value **362**  
 definition-variable **362**  
 definition? **362**  
 deKleer Johan 407  
 dekompozycja programu na części 25  
 delay (forma specjalna) **314**  
     dlaczego forma specjalna 314  
     implementacja za pomocą lambda-abstrakcji 317  
     jawne użycie 341  
     leniwe obliczanie i - 400  
     ze spamiętywaniem 317, 325  
 delay-it **396**  
 delete-queue! 256, **259**  
 denom 83, 86  
     aksjomat opisujący 90  
     skracanie ułamków **89**  
 deposit (operacja na koncie bankowym) **219**  
     z zewnętrznym szeregowaniem **302**  
 deriv  
     różniczkowanie numeryczne **73**  
     różniczkowanie symboliczne **146**  
     sterowane danymi **182**  
 deskryptor łącznika 561  
 diagram  
     przebiegu zdarzeń 293  
     przepływu sygnałów 113, 338  
 Dijkstra Edsger Wybe 304  
 Dinesman Howard P. 409  
 Diofantos 50  
 disjoin **463**  
 display (procedura pierwotna) 52, **86**  
     display-line **313**  
     display-stream **313**  
 distinct? **410**  
 div (procedura ogólna) **186**  
 div-complex **172**

- equal?** 143  
**Eratostenes** 320  
**error** (procedura pierwotna) 67  
**Escher** Maurits Cornelis 126  
**estimate-pi** 223  
**Euklides**  
 algorytm 47–48, 484  
 dla wielomianów 209  
 złożoność 48  
 dowód istnienia nieskończoność wielu  
 liczb pierwszych 324  
*Elementy* 47  
**Euler** Leonhard 70  
 akcelerator szeregiów 329  
 dowód małego twierdzenia Fermata  
 49  
**euler-transform** 329  
**ev-application** 541  
**ev-assignment** 549  
**ev-begin** 545  
**ev-definition** 550  
**ev-if** 549  
**ev-lambda** 540  
**ev-quoted** 540  
**ev-self-eval** 540  
**ev-sequence**  
 bez rekursji ogonowej 548  
 z rekursją ogonową 546  
**ev-variable** 540  
**eval** (leniwe) 393  
**eval** (metacykliczne) 356, 357  
 pierwotne eval a - 378  
 sterowane danymi 366  
 z fazą analizy 386  
**eval** (procedura pierwotna) 378  
 MIT Scheme 378  
 użycie w interpreterze zapytań 464  
**eval-assignment** 359  
**eval-definition** 360  
**eval-dispatch** 540  
**eval-if** (leniwe) 395  
**eval-if** (metacykliczne) 359  
**eval-sequence** 359  
**even-fibs** 113, 116  
**even?** 44  
**evaluator** 352  
 jako maszyna abstrakcyjna 376  
 jako maszyna uniwersalna 376  
 leniwy 390–400
- evaluator**  
 metacykliczny 354  
 niedeterministyczny 418–429  
 z fazą analizy 385–389  
 jako podstawa evaluatora  
 niedeterministycznego 418  
**let** 389  
 z rekursją ogonową 547  
 zapytań 453  
**evaluator** języka Scheme  
 metacykliczny  
 abstrakcja danych w - 355, 356,  
 368, 372  
 cykl eval– apply 355, 356  
 efektywność 385  
**eval** i **apply** 356–360  
**eval**, sterowane danymi 366  
 formy specjalne (dodatkowe)  
 366–368  
 formy specjalne jako wyrażenia  
 pochodne 364–365  
**język** implementowany a język  
 implementacji 359  
 kolejność obliczania argumentów  
 360  
 kombinacje (stosowanie procedur)  
 365  
 kompilacja 601  
 operacje na środowiskach 369  
 pętla sterująca 374  
 procedury pierwotne 373–374  
 procedury wyższych rzędów w -  
 358  
 reprezentacja prawdy i fałszu 368  
 reprezentacja procedur 369  
 reprezentacja środowisk 370–372  
 reprezentacja wyrażeń 360–365  
 reprezentacja wyrażeń w - 356  
 różniczkowanie symboliczne i -  
 360  
 składnia obliczanego języka  
 360–366, 368  
 środowisko globalne 373  
 środowiskowy model obliczeń 354  
**true** i **false** 373  
 uruchamianie 372–376  
 wyrażenia pochodne 364–365  
 z fazą analizy 385–389  
 zadanie - 355

- ewaluator języka Scheme  
z jawnie określonym sterowaniem 537–556  
ciągi wyrażeń 545–548  
definicje 550  
formy specjalne (dodatkowe) 550  
instrukcje warunkowe 548  
jako maszyna uniwersalna 556  
jako program w języku  
    maszynowym 557  
kombinacje 541–545  
model maszyny 552  
monitorowanie wydajności  
    (wykorzystanie stosu) 553–556  
normalna kolejność obliczania 551  
obliczanie argumentów 542–544  
obsługa błędów 552, 556  
operacje 538  
optymalizacje (dodatkowe) 564  
pętla sterująca 551  
procedury pierwotne 544  
procedury złożone 545  
przypisania 549  
rejestry 539  
rekursja ogonowa 547–548, 554,  
    555  
sterownik 539–552  
stosowanie procedur 541–545  
ścieżki danych 538–539  
uruchamianie 551–553  
użycie stosu 541  
wyrażenia nie zawierające  
    podwyrażeń wymagających  
    obliczania 540–541  
wyrażenia pochodne 550  
zmodyfikowany pod kątem  
    skompilowanego kodu 594–596
- exchange** 301  
**execute** 509  
**execute-application**  
    metacykliczne 388  
    niedeterministyczne 425  
**exp** (rejestr) 539  
**expand-clauses** 365  
**expmod** 50, 53, 54  
**expt**  
    obliczane za pomocą maszyny  
        rejestrowej 503  
    z iteracją liniową 43
- expt**  
    z rekursją liniową 43  
**extend-environment** 369, 370  
**extend-if-consistent** 466  
**extend-if-possible** 470  
**external-entry** 596  
**extract-labels** 512
- #f** 17
- factorial**  
    jako maszyna abstrakcyjna 376  
    kompilacja 581–586  
    maszyny rejestrowej 502  
    obliczane (iteracyjnie) za pomocą  
        maszyny rejestrowej 487, 490  
    obliczane (rekurencyjnie) za pomocą  
        maszyny rejestrowej 498–501  
    struktura środowisk w trakcie  
        obliczania 238  
    wykorzystanie stosu 554, 555  
        skompilowany kod 599  
    z iteracją liniową 32  
    z procedurami wyższych rzędów 59  
    z przypisaniem 230  
    z rekursją liniową 31  
    zużycie stosu maszyny rejestrowej  
        523
- false** 17  
**false?** 368  
**fałsz** 17
- fast-expt** 44  
**fast-prime?** 51  
**Feeley** Marc 385  
**Fenichel** Robert 532  
**Fermat** Pierre de 49  
    małe twierdzenie Fermata 49
- fermat-test** 51  
**fetch-assertions** 472  
**fetch-rules** 472  
**fib**  
    przy użyciu nazwanego let 368  
    rozgałęzająca się rekurencyjnie 36  
    wersja liniowa, iteracyjna 38  
    wersja logarytmiczna 46  
    wersja rozgałęzająca się  
        rekurencyjnie 555  
    obliczana za pomocą maszyny  
        rejestrowej 503, 504

- fib**  
 wykorzystanie stosu 555  
 skompilowany kod 599  
 ze spamiętywaniem 266
- fibs** (strumień nieskończony) 320  
 definicja uwikłana 322
- filter** 115
- filtered-accumulate** 60
- filtr** 60, 114
- find-assertions** 465
- find-divisor** 49
- first-agenda-item** 274, 279
- first-exp** 363
- first-frame** 370
- first-operand** 364
- first-segment** 277
- first-term** 202, 206
- fixed-point** 68  
 metodą kolejnych przybliżeń 77
- fixed-point-of-transform** 74
- flag** (rejestr) 509
- flatmap** 122
- flatten-stream** 475
- flip-horiz** 128, 139
- flip-vert** 128, 137
- flipped-pairs** 130, 132
- Floyd Robert 407
- fold-left** 120
- fold-right** 120
- for-each** 107, 399
- for-each-except** 287
- Forbus Kenneth D. 407
- force** 314, 317  
 porównanie z wymuszaniem  
 obliczenia domknięcia 393
- force-it** 396  
 ze spamiętywaniem 397
- forget-value!** 283, 288
- forma specjalna 10  
 porównanie z procedurą 392, 400  
 wyrażenia pochodne w evaluatorze  
 364
- formatowanie wprowadzanych  
 wyrażeń 7
- formy specjalne (oznaczone przez *ns*)  
 nie wchodzą w skład standardu  
 IEEE języka Scheme)
- and 18
- begin** 216
- formy specjalne  
**cond** 16
- cons-stream** (*ns*) 314
- define** 7, 12
- delay** (*ns*) 314
- if** 18
- lambda** 61
- let** 63
- let\*** 367
- letrec** 383
- nazwany let 367
- or** 19
- quote** 142
- set!** 216
- Fortran 3, 118, 349
- ograniczenia dotyczące danych  
 złożonych 98
- frame-coord-map** 134
- frame-values** 370
- frame-variables** 370
- Franz Lisp 3
- free** (rejestr) 529, 533
- Friedman Daniel P. 318, 353
- fringe** 110  
 wyliczenie liści drzewa 116
- front-ptr** 258
- front-queue** 256, 258
- funkcja (matematyczna)  
 $\mapsto$  na oznaczenie - 68
- Ackermann 35
- pochodna 73
- porównanie z procedurami 21–22
- punkt stały 67–69
- wyglądzanie 77
- wymierna 208–212  
 skracanie 211–212
- złożenie 76
- wielokrotne 76
- funkcyjny język programowania 349
- gcd** 47  
 realizacja w postaci maszyny  
 rejestrowej 506
- gcd-terms** 210
- generate-huffman-tree** 166
- generator kodu 560
- argumenty 561
- wynik działania 562

- generator liczb losowych 216, 221  
metoda Monte Carlo 222  
w teście pierwszości 50  
z ustawianiem 225  
z ustawianiem, wersja strumieniowa 347  
generowanie zdań 417  
**get** 179, 265  
**get-contents** 508  
**get-global-environment** 551  
**get-register** 511  
**get-register-contents** 506, 511  
**get-signal** 270, 273  
**get-value** 283, 288  
Goguen Joseph 91  
**goto** (instrukcja maszyny rejestrowej) 488  
rejestr jako punkt docelowy 497  
symulacja 517  
**goto-dest** 518  
gramatyka 412  
Gray Jim 308  
Griss Martin Lewis 3  
Guttag John Vogel 91
- Hamming Richard Wesley 162, 324  
Hardy Godfrey Harold 324, 336  
**has-value?** 283, 288  
hasło, zabezpieczenie konta bankowego 221  
Havender J. 308  
Haynes Christopher T. 353  
Hearn Anthony C. 3  
Henderson Peter 126, 321  
diagram Hendersona 321  
Heraklit 213  
Heron z Aleksandrii 22  
Hewitt Carl Eddie 34, 407, 430, 532  
hierarchia typów 194–199  
niedostatki 196  
w algebrze symbolicznej 207–208  
hierarchiczne struktury danych 97, 107–111  
Hilfinger Paul 158  
hipoteza Bertranda 324  
Hoare Charles Antony Richard 90  
Hodges Andrew 377  
Hofstadter Douglas R. 378
- Horner W. G. 118  
schemat Hornera 118  
Huffman David 161  
Hughes R. J. M. 402
- IBM 704 85  
**identity** 57  
**if** (forma specjalna) 18  
dlaczego potrzebna jest forma specjalna 24  
jednoczłonowy (bez alternatywy) 278  
normalna kolejność obliczania 21  
obliczanie wartości 18  
porównanie z **cond** 18  
predykat, następnik i alternatywa 18  
**if-alternative** 363  
**if-consequent** 363  
**if-predicate** 363  
**if?** 363  
**iloczyn** logiczny 268  
**imag-part**  
dla danych ze znacznikami typu 176  
programowanie sterowane danymi 181  
reprezentacja biegunowa liczb zespolonych 173  
reprezentacja prostokątna liczb zespolonych 172  
**imag-part-polar** 175  
**imag-part-rectangular** 174  
**inc** 57  
indeks wektora 526  
indeksowanie bazy danych 446, 471  
**inform-about-no-value** 284  
**inform-about-value** 284  
informatyka 353, 376  
porównanie z matematyką 21, 429  
**initialize-stack** (operacja maszyny rejestrowej) 509, 522  
**insert!**  
dla tablicy dwuwymiarowej 264  
dla tablicy jednowymiarowej 262  
**insert-queue!** 256, 258  
**install-complex-package** 188  
**install-polar-package** 180  
**install-polynomial-package** 201  
**install-rational-package** 187

- 
- install-rectangular-package 179
  - install-scheme-number-package 187
  - instantiate 461
  - instruction-execution-proc 514
  - instruction-text 513
  - instrukcja maszyny rejestrowej 483
  - integers (strumień nieskończony) 320
    - definicja uwikłana 322
    - z leniwymi listami 401
  - integers-starting-from 319
  - integral 58, 337, 342
    - z lambda-abstrakcją 61
    - z leniwymi listami 402
    - z odroczonym argumentem 341
  - integrate-series 326
  - interfejs składniowy 273
  - interleave 334
  - interleave-delayed 474
  - Interlisp 3
  - interpreter 2, 352
    - języka Scheme napisany w C 601
    - pętla wczytaj-oblicz-wypisz 7
    - porównanie z kompilatorem 557–558, 597
    - zapytań
      - baza danych 471–474
      - dopasowywanie wzorców 445–446, 465–467
      - evaluator zapytań 453, 461–464
      - operacje na strumieniach 474–475
      - opis ogólny 445–454
      - pętla sterująca 453, 460–461
      - podstawianie 461
      - porównanie z interpreterem Lispu 452, 453, 481
      - problemy z not i lisp-value 457–458, 481
      - ramka 445, 478
      - reprezentacja zmiennych we wzorcach 461, 476–478
      - składnia języka zapytań 475–478
      - struktura środowisk 482
      - strumień ramek 446, 453
      - ulepszenia 459, 481
      - unifikacja 450–451, 468–471
  - interpreter
    - zapytań
      - wprowadzanie asercji i reguł 453
      - zapętlenie 455–456, 459
  - intersection-set 150
    - reprezentacja zbiorów za pomocą drzew binarnych 158
    - reprezentacja zbiorów za pomocą list nieuporządkowanych 151
    - reprezentacja zbiorów za pomocą list uporządkowanych 153
  - inverter 271
  - inwerter 267, 271
  - inżynieria, porównanie z matematyką 51
  - iteracja, zaimplementowana poprzez wywołanie procedury 24, 34
  - iteracyjny proces obliczeniowy 33
    - konstruowanie algorytmów 45
    - liniowy 33, 42
  - jednoznakowa makrodefinicja wczytywania 477
  - język
    - bardzo wysokiego poziomu 22
    - graficzny 126–140
    - maszynowy 557
      - a język wysokiego poziomu 351
      - naturalny, cytowanie 141
      - opisu maszyn rejestrowych 487–491
        - assign 489, 504
        - branch 488, 505
        - const 489, 504, 505
        - etykiety 488
        - goto 488, 505
        - instrukcje 488, 504
        - label 488, 505
        - op 489, 505
        - perform 491, 505
        - print 491
        - punkt wejścia 488
        - read 490
        - reg 489, 504
        - restore 500, 505
        - save 500, 505
        - test 488, 505
      - programowania 1
        - bardzo wysokiego poziomu 22
        - funkcyjny 349

- język**  
programowania  
obiektowy 197  
projekt 390  
w logice 431  
ze ścisłą kontrolą typów 344  
przenoszenie 598  
rodzimy maszyny 557  
wysokiego poziomu, a język  
maszynowy 351  
zapytań 432–444  
bazy danych 433–435  
dedukcja logiczna 442–444  
metody abstrakcji 439  
porównanie z logiką matematyczną  
454–459  
rozszerzenia 458, 480  
źródłowy 557
- Kaldewajj Anne 46  
kalkulator, obliczanie punktów stałych  
68  
Karr Alphonse 213  
katalogowanie symboli 528  
Kepler Johannes 483  
key 159  
Khayyam Omar 41  
klauzula  
hornowska 430  
w wyrażeniu warunkowym 17  
wyrażenia cond, dodatkowa  
składnia 366  
klucz rekordu  
porównywanie 265  
w bazie danych 158  
w tablicy 261  
Knuth Donald E. 41, 45, 47, 118  
kod  
ASCII 159  
Huffmana 159–167  
optymalność 162  
złożoność kodowania 167  
Morse'a 160  
prefiksowy 160  
rozdzielający 160  
stałej długości 160  
zmiennej długości 160  
kodowanie otwarte operacji  
pierwotnych 589, 594
- kolejka 256–260  
dwukierunkowa 260  
FIFO 256  
implementacja proceduralna 260  
koniec 256  
operacje 256  
początek 256  
użyta w kolejce zdarzeń symulacji  
276  
zdarzeń 274
- kolejność obliczania  
argumentów  
w evaluatorze  
niedeterministycznym 417  
w metacyklicznym evaluatorze  
języka Scheme 360  
języka Scheme 231  
normalna 16  
porównanie normalnej i stosowanej  
20  
przypisanie i - 231  
stosowana 16  
w evaluatorze metacyklicznym 360  
w evaluatorze z jawnie określonym  
sterowaniem 543  
w kompilatorze 585  
zależna od implementacji 233
- kolejność zdarzeń  
nieokreśloność w systemach  
współbieżnych 292  
struktura procedur a - 317  
Kolmogorow A. N. 222  
kombinacja 5–7  
jako drzewo 9  
jako operator kombinacji 71  
lambda-abstrakcja jako operator 61  
obliczanie wartości 8–10  
ze złożonym wyrażeniem jako  
operatorem 20  
kombinacje zagnieżdżone 6–7  
komentarze w programach 123  
komórka, w implementacji semafora  
305  
kompatybilność wsteczna 399  
kompilator 556–558  
języka Scheme 558–601  
adresowanie składniowe 590–592  
ciąg instrukcji 578–581  
ciągi wyrażeń 569

- kompilator  
 języka Scheme  
     cytowanie 565  
     definicje 566  
     generowanie etykiet 567  
     kod łączników 565  
     kodowanie otwartej operacji  
         pierwotnych 589, 594  
     kolejność obliczania 585  
     kombinacje 571–578  
     lambda-wyrażenia 569  
     porównanie z evaluatorem z fazą  
         analizy 559, 560  
     porównanie z evaluatorem  
         z jawnie określonym  
             sterowaniem 558–560, 564, 597  
     procedury składni wyrażeń 560  
     przykład komplikacji 581–584  
     przypisania 566  
     rekursja ogonowa 576  
     samoobliczające się wyrażenia 565  
     scalanie z evaluatorem 594–601  
     stosowanie procedur 571–578  
     struktura - 560–564  
     uruchamianie skompilowanego  
         kodu 594–601  
     używane operacje maszynowe 558  
     używane rejesty 557, 577  
     wydajność 559–560  
         wykorzystanie stosu, 562, 564,  
             585, 597, 599  
     wyłuskiwanie definicji  
         wewnętrznych 592, 593  
     wyrażenia warunkowe 567  
     zmienne 565  
     porównanie z interpreterem  
         557–558, 597  
     rekursja ogonowa, przydział pamięci  
         na stosie i odśmiecanie 577  
 komputer  
     analogowy 341  
     ogólnego przeznaczenia  
         jako maszyna uniwersalna 557  
 konkretna reprezentacja danych 83  
 konstrukcje pętli 34  
 konstruktor 83  
     jako bariera abstrakcji 88  
     konto bankowe 215, 245  
     implementacja z sekcją krytyczną  
         298  
     model strumieniowy 347  
     przelew 303  
     wspólne 228, 231  
         z równoległym dostępem 292  
     zamodelowane za pomocą  
         strumieni 350  
     zabezpieczone hasłem 221  
     zamiana sald 301  
 kontynuacja  
     niepowodzenia (evaluator  
         niedeterministyczny) 418, 420  
     tworzona przez amb 426  
     tworzona przez pętlę sterującą 426  
     tworzona przez przypisanie 423  
     udanego obliczenia (evaluator  
         niedeterministyczny) 418, 420  
     w evaluatorze niedeterministycznym  
         418–420  
     w symulatorze maszyny rejestrowej  
         513  
 konwencja nazywania  
     ! dla przypisań i zmian 216  
     ? w nazwach predykatów 23  
 konwencjonalny interfejs 82  
     ciągi jako - 112–125  
 Kowalski Robert 430  
 KRC 121, 334  
 kropka dziesiętna w liczbach 23  
 kryptografia 52  
 kształt procesu obliczeniowego 33  
 kwadrat jednostkowy 133  
 kwantowanie czasu 306  
  
 label (instrukcja maszyny  
     rejestrowej) 488  
     symulacja 519  
 label-exp 519  
 label-exp-label 519  
 Lagrange'a wzór interpolacyjny 200  
 lambda (forma specjalna) 61  
     notacja kropki i ogona 104  
     porównanie z define 61  
 lambda-abstrakcja, jako operator  
     kombinacji 61  
 lambda-body 362  
 lambda-parameters 362

- lambda?** 362  
**lambda-wyrażenie**, wartość 235  
**Lambert J. H.** 71  
**Lamé Gabriel** 48  
**Lamport Leslie** 309  
**Landin Peter** 11, 318  
**Last-exp?** 363  
**last-operand?** 542  
**last-pair** 102, 250  
reguły 444  
**leaf?** 163  
**left-branch** 155, 164  
**Leibniz Baron Gottfried Wilhelm von**  
dowód małego twierdzenia Fermata  
49  
szereg zbieżny do  $\frac{\pi}{4}$  56  
szereg zbieżny do  $\pi$  328  
**Leiserson Charles E.** 156, 336  
**length** 101  
wersja iteracyjna 101  
wersja rekurencyjna 101  
za pomocą kumulacji 118  
**leniwa lista** 400–403  
**leniwa para** 400–403  
**leniwe drzewo** 402  
**leniwe obliczanie** 390  
**let** (forma specjalna) 63  
lukier syntaktyczny 64, 243  
model obliczeniowy 243  
nazwany 367  
porównanie z definicjami lokalnymi  
64  
zakres zmiennych 64  
**let\*** (forma specjalna) 367  
**letrec** (forma specjalna) 383  
**lexical-address-lookup** 591, 592  
**lexical-address-set!** 591, 592  
**liczba podpiersiastkowa** 22  
**liczby**  
całkowite 5, 23  
dzielenie 23  
całkowite a rzeczywiste 5  
**Carmichaela** 51, 54  
**Churcha** 93  
**Fibonacciego** 36  
a algorytm Euklidesa obliczania  
NWD 48  
**kropka dziesiętna** 23  
**losowe, generator** 221  
**liczby**  
ogólny system arytmetyczny 186  
pierwsze 48–52  
a kryptografia 52  
sito Eratostenesa 320  
test Fermata 49–52  
test Millera-Rabina 54  
test pierwszości 48–55  
**porównanie** 17  
**pseudolosowe** 221  
**Ramanujana** 336  
**równość** 17, 144, 527  
**rzeczywiste** 5  
w **Lispie** 5  
**wymierne** 23  
ogólny system arytmetyczny 187  
operacje arytmetyczne 83–87  
reprezentowane jako pary 85  
skracanie ułamków 87, 88  
w **MIT Scheme** 23  
wypisywanie 86  
względnie pierwsze 60  
zależne od implementacji 23  
**zespolone**  
arytmetyka 169  
reprezentacja biegunowa 170, 173  
reprezentacja prostokątna 170, 172  
reprezentacja ze znacznikami  
173–177  
**licznik rozkazów** 509  
**Lieberman Henry** 532  
**linia świata cząstki** 310, 348  
**liniowy iteracyjny proces**  
obliczeniowy 33  
złożoność 42  
**liniowy rekurencyjny proces**  
obliczeniowy 33  
złożoność 42  
**Liskov Barbara Huberman** 91  
**Lisp**  
akronim **LISt Przetwarzania** 2  
dialekty  
Common Lisp 3  
Franz Lisp 3  
Interlisp 3  
MacLisp 3  
MDL 533  
Portable Standard Lisp 3

- Lisp  
 dialekty  
   Scheme 3  
   Zetalisp 3  
 efektywność 3  
 historia 2, 3  
 na komputer DEC PDP-1 532  
 odpowiedniość do pisania  
   ewaluatorów 353  
 pierwsza implementacja na  
   komputer IBM 704 85  
 porównanie z Fortranem 3  
 porównanie z Pascalem 11  
 procedury pierwszorzędne 75  
 stosowana kolejność obliczania 16  
 wewnętrzny system typów 190  
 wydajność 7  
 wyjątkowe cechy 4  
**lisp-value** (interpretator zapytań) 464  
**lisp-value** (w języku zapytań) 439,  
   457  
 obliczanie 449, 464, 481  
**list** (procedura pierwotna) 99  
**list->tree** 158  
**list-difference** 579  
**list-of-arg-values** 394  
**list-of-delayed-args** 395  
**list-of-values** 359  
**list-ref** 101, 401  
**list-union** 579  
**lista** 99  
   cdrowanie 100  
   konstruowanie 102  
   cytowanie 142  
   długość 101  
   leniwa 400–403  
   łączenie za pomocą append 102  
   nieuporządkowana, reprezentacja  
     zbiorów 150–152  
   n-ty element 100  
   odwracanie 102  
   odwzorowania 104–107  
   operacje 100–104  
   operacje car, cdr i cons 100  
   ostatnia para 102  
   przekształcenie drzewa binarnego  
     w - 157  
   przekształcenie w drzewo binarne  
     157
- lista  
 pusta 100  
 oznaczana jako ' () 142  
 rozpoznawanie za pomocą null?  
   101  
 równość 143  
 sposób wypisywania 99  
 techniki przetwarzania 100–104  
 uporządkowana, reprezentacja  
   zbiorów 152–154  
 wolnych indeksów 529  
 wsteczny apostrof 565  
 wyrazów wielomianu, reprezentacja  
   204–206  
 z atrapą 261, 277  
 z cyklem 251  
 liść drzewa 9  
**lives-near** (reguła) 439, 442  
 Locke John 1  
**log** (procedura pierwotna) 69  
 logarytm, przybliżenie ln 2 332  
**logical-not** 271  
 lokacja 525  
 lokalne zmienne stanu 215–221  
 lokalny rozwój procesu  
   obliczeniowego 30  
**lookup**  
   dla tablicy dwuwymiarowej 263  
   dla tablicy jednowymiarowej 262  
   zbiór rekordów 159  
**lookup-label** 514  
**lookup-prim** 520  
**lookup-variable-value** 369, 371  
   dla definicji wyłuskanych 381  
**lower-bound** 94  
 lukier syntaktyczny 11  
**define** 362  
 konstrukcje iteracyjne jako - 34  
**let** jako - 64  
 procedury i dane 273
- łamigłówki logiczne 409–412  
 łączenie, sposoby 4  
 łącznik 561
- macierz, reprezentowana jako ciąg  
   119  
**Macintosh** 556  
**MacLisp** 3

- magnitude**  
dla danych ze znacznikami typu **176**  
programowanie sterowane danymi  
**181**  
reprezentacja biegunowa liczb  
zespolonych **173**  
reprezentacja prostokątna liczb  
zespolonych **172**  
**magnitude-polar** **175**  
**magnitude-rectangular** **174**  
**make-account** **219**  
w modelu środowiskowym **245**  
z sekcją krytyczną **298, 300**  
**make-account-and-serializer**  
**302**  
**make-accumulator** **220**  
**make-agenda** **274, 277**  
**make-assign** **515**  
**make-begin** **363**  
**make-branch** **517**  
**make-center-percent** **95**  
**make-center-width** **95**  
**make-code-tree** **163**  
**make-compiled-procedure** **570**  
**make-complex-from-mag-ang** **189**  
**make-complex-from-real-imag**  
**189**  
**make-connector** **286**  
**make-cycle** **251**  
**make-decrementer** **226**  
**make-execution-procedure** **515**  
**make-frame** **133, 135, 370**  
**make-from-mag-ang** **176, 182**  
przekazywanie komunikatów **185**  
reprezentacja biegunowa liczb  
zespolonych **173**  
reprezentacja prostokątna liczb  
zespolonych **172**  
**make-from-mag-ang-polar** **175**  
**make-from-mag-ang-rectangular**  
**175**  
**make-from-real-imag** **176, 182**  
przekazywanie komunikatów **184**  
reprezentacja biegunowa liczb  
zespolonych **173**  
reprezentacja prostokątna liczb  
zespolonych **172**  
**make-from-real-imag-polar** **175**
- make-from-real-imag-rectangular**  
**175**  
**make-goto** **517**  
**make-if** **363**  
**make-instruction** **513**  
**make-instruction-sequence** **563**  
**make-interval** **93, 94**  
**make-joint** **231**  
**make-label** **568**  
**make-label-entry** **514**  
**make-lambda** **362**  
**make-leaf** **163**  
**make-leaf-set** **165**  
**make-machine** **505, 507**  
**make-monitored** **220**  
**make-mutex** **305**  
**make-new-machine** **510**  
**make-operation-exp** **519**  
**make-perform** **518**  
**make-point** **89**  
**make-poly** **201**  
**make-polynomial** **206**  
**make-primitive-exp** **519**  
**make-procedure** **369**  
**make-product** **146, 148**  
**make-queue** **256, 258**  
**make-rat** **83, 86, 88**  
aksjomat opisujący **90**  
skracanie ułamków **87**  
**make-rational** **188**  
**make-register** **507**  
**make-restore** **518**  
**make-save** **518**  
**make-scheme-number** **187**  
**make-segment** **89, 136**  
**make-serializer** **305**  
**make-simplified-withdraw** **225, 348**  
**make-stack** **508**  
z monitorowaniem stosu **522**  
**make-sum** **146, 148**  
**make-table**  
dla tablicy jednowymiarowej **262**  
implementacja za pomocą  
przekazywania komunikatów **264**  
**make-tableau** **330**  
**make-term** **202, 206**  
**make-test** **516**  
**make-time-segment** **277**

**make-tree** 155  
**make-vect** 135  
**make-wire** 268, 272, 276  
**make-withdraw** 218  
 w modelu środowiskowym 239–243  
 z użyciem **let** 243  
**makrodefinicja** 365  
 wczytywania, jednoznakowa 477  
**małe twierdzenie Fermata** 49, 50  
 dowód 49  
 postać alternatywna 54  
**map** 105, 401  
 o wielu argumentach 105  
 za pomocą kumulacji 118  
**map-over-symbols** 477  
**map-successive-pairs** 346  
**maszyna**  
 rejestrów 483  
 akcje 490–491  
**assign** 489, 504  
**branch** 488, 505  
**const** 489, 504, 505  
 diagram sterownika 486  
 diagram ścieżek danych 485  
 etykiety 488  
**goto** 488, 505  
 instrukcje 483, 488, 504  
 język opisu 487–491  
**label** 488, 505  
**op** 489, 505  
 operacje 484–487  
**perform** 491, 505  
 podprogram 494–498  
**print** 491  
 projektowanie 484–505  
 punkty wejścia 488  
**read** 490  
**reg** 489, 504  
**restore** 500, 505  
**save** 500, 505  
 sprawdzanie warunków 486  
**sterownik** 484–487  
**stos** 498–504  
**symulator** 505–524  
 ścieżki danych 484–487  
**test** 488, 505  
 wydajność 522–524

**maszyna**  
 Turinga 377  
 uniwersalna 376  
 evaluator z jawnie określonym sterowaniem 556  
 jako komputer ogólnego przeznaczenia 557  
**matematyka**  
 porównanie z informatyką 21, 429  
 porównanie z inżynierią 51  
**matrix--matrix** 120  
**matrix--vector** 120  
**max** (procedura pierwotna) 94  
 McAllester David Allen 408  
 McCarthy John 2, 405  
 McDermott Drew 407  
**MDL** 533  
 mechanika kwantowa 350  
**member** 410  
**memo-fib** 266  
**memo-proc** 318  
**memoize** 266  
**memq** 143  
**merge** 325  
**merge-weighted** 335  
 metacykliczny evaluator 354  
 języka Scheme 354–379  
 brak określenia rekursji ogonowej 547  
**metoda**  
 bisekcji 65–67  
**half-interval-method** 67  
 porównanie z metodą Newtona 73  
 kolejnych przybliżeń 77  
 Monte Carlo 222  
 całkowanie 224  
 całkowanie, sformułowanie strumieniowe 347  
 sformułowanie strumieniowe 345  
**Newtona**  
 dla funkcji różniczkowalnych 72–74  
 obliczania pierwiastków sześciennych 25  
 pierwiastkowania 21–24, 74  
 porównanie z metodą bisekcji 73  
 Simpsona całkowania numerycznego 59

- miara, w pierścieniu euklidesowym 209  
MicroPlanner 407  
*midpoint-segment* 89  
Mikrusoft 433  
Miller Gary L. 54  
*min* (procedura pierwotna) 94  
minimalne zobowiązania, zasada 173  
Minsky Marvin Lee xxi, 532  
Miranda 121  
MIT  
    Artificial Intelligence Laboratory 3  
    Project MAC 3  
    Research Laboratory of Electronics 2, 532  
    wczesna historia 126  
MIT Scheme  
    definicje wewnętrzne 381  
    *eval* 378  
    liczby 23  
    pusty strumień 312  
    *random* 225  
    *user-initial-environment* 378  
    *without-interrupts* 306  
ML 345  
model podstawieniowy  
    obliczeń 232  
    stosowania procedur 13–16  
modele  
    danych, abstrakcyjne 90  
    obliczeń 551  
modelowanie  
    strategia projektowa 213  
    w nauce i inżynierii 14  
*modifies-register?* 578  
modularność 117, 213  
modelowanie za pomocą obiektów 221  
programy funkcyjne a obiekty 345–350  
przekierowanie ze względu na typ 177  
strumienie i – 327  
zasada ukrywania informacji 217  
zastosowanie strumieni 347  
zgodna z granicami obiektów 350  
modulo  $n$  50  
*modus ponens* 454  
modyfikatory 246  
modyfikowalne obiekty danych 246–255  
implementacja za pomocą przypisań 254–255  
pary 247–251  
reprezentacja proceduralna 254–255  
struktury listowe 247–251  
współdzielenie danych 253  
monitorowanie procedur 220  
Monte Carlo, metoda 222  
*monte-carlo* 223  
    strumień nieskończony 346  
Moon David A. 3, 532  
Morse'a kod 160  
*mul* (procedura ogólna) 186  
    mnożenie współczynników wielomianów 203  
*mul-complex* 171  
*mul-interval* 94  
    bardziej efektywna wersja 95  
*mul-poly* 201  
*mul-rat* 84  
*mul-series* 326  
*mul-streams* 324  
*mul-terms* 203  
Multics, system z podziałem czasu 532  
*multiple-dwelling* 410  
*multiplicand* 147  
*multiplier*  
    selektor 147  
    więzy pierwotne 285  
Munro Ian 119  
*mystery* 251  
nadawanie nazw procedurom 12  
nadtypy 194  
    wiele – 196  
największy wspólny dzielnik 47–48  
    obliczany za pomocą maszyny rejestrowej 484–486  
operacja ogólna 210  
przybliżenie  $\pi$  222  
wielomianów 209–211  
zastosowanie w arytmetyce liczb wymiernych 87  
napis  
    cytowanie 142  
operacje pierwotne 477, 568

- następnik  
 klauzuli 17  
 wyrażenia warunkowego **if** 18  
 nawiasy  
     ograniczające klauzule w wyrażeniu warunkowym 17  
     ograniczające kombinacje 6  
     w definicji procedury 12  
 nawracanie, w przeszukiwaniu 407  
 nazwa  
     lokalna 27–28, 62–65  
     procedury 12  
 nazwany **let** (forma specjalna) **367**  
 nazywanie  
     obiektów obliczeniowych 7  
     parametrów formalnych 27  
**needs-register?** **578**  
**negate** **463**  
**new** (rejestr) **535**  
**new-cars** (rejestr) **533**  
**new-cdrs** (rejestr) **533**  
**new-withdraw** **217**  
**newline** (procedura pierwotna) **52**, **86**  
**newton-transform** **73**  
**newtons-method** **73**  
**next** (deskryptor łącznika) **561**  
**next-to** (reguła) **444**  
 niedeterminizm, w działaniu  
     programów współbieżnych 295, 350  
 niepowodzenie, w obliczeniach  
     niedeterministycznych 405  
 porównanie z błędami 421  
     przeszukiwanie i - 406  
 niewrażliwość na odniesienia 228  
 niezmiennik procesu iteracyjnego 45  
**nil**  
     jak się obyć bez 142  
     jako pusta lista 100  
     jako zwykła zmienna w Scheme 100  
     znacznik końca listy 99  
**no-more-exp?** **547**  
**no-operands?** **364**  
 normalna kolejność obliczania 16  
     **if** 21  
     obliczenia odroczone i - 344–345
- normalna kolejność obliczania  
 porównanie ze stosowaną kolejnością obliczania 20, 48, 390–392  
 w evaluatorze z jawnie określonym sterowaniem 551  
**not** (procedura pierwotna) **19**  
**not** (w języku zapytań) 438, 457  
     obliczanie 448, 463, 481  
 notacja  
      $\mapsto$ , na oznaczenie funkcji matematycznych 68  
     infiksowa a notacja prefiksowa 149  
     kropki i ogona  
         dla parametrów procedur 103, 181  
         **read** 467  
         w regule języka zapytań 443  
         w zapytaniu 436  
         we wzorcach zapytań 467  
         we wzorcu 436  
     prefiksowa 6  
         a notacja infiksowa 149  
     pudełkowo-wskaźnikowa 97  
         znacznik końca listy 99  
     rzędów wielkości 42  
      $\Sigma$ , notacja sigma 57  
**nouns** **412**  
**null?** (procedura pierwotna) **101**  
 implementacja przy użyciu  
     wskaźników ze znacznikiem typu 530  
**number?** (procedura pierwotna) **145**  
 implementacja przy użyciu  
     wskaźników ze znacznikiem typu 530  
 typy danych 190  
**numer** **83, 86**  
     aksjomat opisujący 90  
     skracanie ułamków **89**
- obarray** **528**  
**obiekt(y)** **214**  
     danych, modyfikowalne 246–255  
     korzyści z modelowania za pomocą - 221  
**o stanie zmiennym w czasie** **215**  
**odroczone** **314**  
     współdzielenie danych 253  
**obiektowe języki programowania** **197**

- obliczalność 376  
obliczanie  
and 18, 366  
form specjalnych 10  
if 18  
kombinacji 8–10  
na żądanie 317, 318  
or 19, 366  
spojrzenie z punktu widzenia  
  przetwarzania sygnałów 113  
wyrażeń pierwotnych 9  
wyrażeń warunkowych cond 17
- obliczenie  
modele 551  
niedeterministyczne 403–417  
niedeterministyczne a w Scheme  
  403  
odroczone 214, 310  
odroczone, w leniwym evaluatorze  
  390–403
- obsługa błędów  
  w evaluatorze z jawnie określonym  
    sterowaniem 552, 556  
  w skompilowanym kodzie 598
- obwód elektroniczny  
  modelowanie za pomocą strumieni  
    337, 342  
  RC 337  
  RLC szeregowy 342
- oczekiwanie aktywne 305
- odcinek  
  reprezentowany jako para punktów  
    89  
  reprezentowany jako para wektorów  
    136
- odpluskwanie 2
- odraczanie obliczeń 214, 310  
  normalna kolejność obliczania i -  
    344–345  
  przypisanie i - 319  
  strumienie i - 339–344  
  wypisywanie i - 318
- odśmiecanie 531–537  
  i rekursja ogonowa 577  
  modyfikowanie danych a - 247  
  przez kopiowanie 532–537  
  przez zaznaczanie i zamiananie 532  
  ze scalaniem 532
- odwzorowywanie  
  drzew 111–112  
  jako przetwornik 114  
  list 104–107  
  zagnieździone 121–125, 332–336
- old (rejestr) 535  
oldcr (rejestr) 536  
ones (strumień nieskończony) 322  
  z leniwymi listami 401
- op (w maszynie rejestrowej) 489  
  symulacja 519
- operacja  
  atomowa  
    implementacja sprzętowa 306  
    test-and-set! 306  
  maszyny rejestrowej 484–487  
  na elementach różnych typów 191  
  odroczone 33  
  ogólna 82
- operacje arytmetyczne, ogólne  
  186–191  
  struktura systemu 186
- operands 364
- operation-exp 520
- operation-exp-op 520
- operation-exp-operands 520
- operator  
  Y 384  
  kombinacji 6  
    będący wyrażeniem złożonym 20  
    kombinacja jako - 71  
    lambda-abstrakcja jako - 61  
  przypisania 215
- operator 364
- opóźnienie, w układzie cyfrowym 267
- optymalność, kodu Huffmmana 162
- opuszczenie semafora 304
- or (forma specjalna) 19  
  bez podwyrażeń 366  
  dlaczego forma specjalna 19  
  obliczanie wartości 19, 366
- or (w języku zapytań) 438  
  obliczanie 448, 463
- order 202, 206
- origin-frame 133
- Ostrowski A. M. 119
- outranked-by (reguła) 441, 458

- pair?** (procedura pierwotna) **109**  
 implementacja przy użyciu  
     wskaźników ze znacznikiem typu  
     530  
**pairs** **334**  
**pakiet** 179  
     arytmetyki języka Scheme 187  
     biegunowa reprezentacja liczb  
         zespolonych 180  
     liczb wymiernych 187  
     liczb zespolonych 188  
     prostokątna reprezentacja liczb  
         zespolonych 179  
     wielomianów 201  
**pamięć**  
     struktur listowych 524–537  
     w 1964 r. 406  
**Pan V. Y.** 119  
**papirus Rhinda** 46  
**parallel-execute** 297  
**parallel-instruction-sequences**  
     **581**  
**parametry formalne** 12  
     nazwy 27  
     zakres 28  
**parse** **414**  
**parse-...** 413–415  
**partial-sums** 324  
**parzyste** 84  
     definicja aksjomatyczna 91  
     leniwe 400–403  
     modyfikowalne 247–251  
     notacja pułapkowo-wskaźnikowa 97  
     reprezentacja ciągów 98  
     reprezentacja drzew 107–111  
     reprezentacja proceduralna 91–92,  
         254–255, 400  
     reprezentacja przy użyciu wektorów  
         526–531  
     strumień nieskończony - 332–336  
**Pascal** 11  
     brak procedur wyższych rzędów 344  
     ograniczenia dotyczące danych  
         złożonych 98  
     procedury rekurencyjne 34  
     słaby punkt w operowaniu  
         obiektami złożonymi 290  
**Pascal Blaise** 41  
     trójkąt Pascala 41  
**pattern-match** **466**  
**pc (rejestr)** 509  
**pełny sumator** 269  
     full-adder **269**  
**perform** (instrukcja maszyny  
     rejestrowej) 491  
**symulacja** 518  
**perform-action** **518**  
**Perlis Alan J.** xix, 98  
     lukier syntaktyczny 11  
     parafraza Oscara Wilde'a 7  
**permutacje zbioru** 123  
**permutations** **123**  
**petla** 24  
     implementacja w evaluatorze  
         metacyklicznym 368  
     sterująca  
         evaluatora leniwego 395  
         evaluatora metacyklicznego  
             Scheme'a 374  
         evaluatora niedeterministycznego  
             408, 426  
         evaluatora z jawnie określonym  
             sterowaniem 551  
         interpretora zapytań 453, 460–461  
         wczytaj-oblicz-wypisz 7  
**Phillips Hubert** 411  
**π (pi)**  
     przybliżenie Cesàro 222, 345  
     przybliżenie metodą bisekcji 67  
     przybliżenie za pomocą całkowania  
         metodą Monte Carlo 224, 347  
     strumień przybliżeń 328–331  
     szereg Leibniza 56, 328  
     wzór Wallisa 59  
**pi-stream** **329**  
**pi-sum** **56**  
     z lambda-abstrakcją **61**  
     z procedurami wyższych rzędów **58**  
**pierścień euklidesowy** 209  
**pierwiastek**  
     czwartego stopnia, jako punkt stały  
         77  
     kwadratowy 21–24  
         jako punkt stały 68  
         strumień przybliżeń 328  
     n-go stopnia, jako punkt stały **77**

- pierwiastek  
sześcienny  
jako punkt stały 72  
liczony metodą Newtona 25
- Pingala Áchárya 45
- Pitman Kent M. 3
- pluskwa 1  
pobożne życzenia 83, 145  
pochodna funkcji 73  
podniesienie semafora 304  
podprogram maszyny rejestrowej  
494–498
- podstawienie zmiennych 437  
pasujące do wzorca 437  
spełniające zapytanie 437
- podstawieniowy model  
obliczeń 232  
sosowania procedur 13–16  
kształt procesu 31–34  
nieadekwatność - 225–227
- podtypy 194  
wiele - 196
- polar (pakiet) 180
- polar? 174
- pole typu 527
- poly, struktura danych 201
- polynomial (pakiet) 201
- pop 508
- poprawność programu 21
- porównywanie liczb 17
- Portable Standard Lisp 3
- postać kanoniczna wielomianów 208
- potęgowanie 43–45  
modulo  $n$  50
- PowerPC 309
- półsumator 268  
half-adder 269  
symulacja 275–276
- prawda 17
- predykat 17
- klauzuli wyrażenia warunkowego  
cond 17
- konwencja nazywania 23  
wyrażenia warunkowego if 18
- prepositions 414
- preserving 562, 564, 580, 585
- prime-sum-pair 403
- prime-sum-pairs 123
- strumień nieskończony 332
- prime? 49, 323
- primes (strumień nieskończony) 321  
definicja uwikłana 323
- primitive-apply 545
- primitive-implementation 373
- primitive-procedure-names 374
- primitive-procedure-objects  
374
- primitive-procedure? 369, 373
- print (operacja maszyny rejestrowej)  
491
- print-point 89
- print-queue 260
- print-rat 86
- print-result 551  
z monitorowanym stosem 553
- print-stack-statistics (operacja  
maszyny rejestrowej) 522
- probe  
w symulacji obwodów cyfrowych  
275
- w systemie więzów 286
- problem  
ośmio hetmanów 124, 412  
stopu 378, 379
- problemy w systemach współbieżnych  
292
- proc (rejestr) 539
- procedura 4  
anonimowa 61  
definicja 12  
definiowanie 11–12  
dowolna liczba argumentów 6, 103  
jako argument 56–60  
jako czarna skrzynka 25–27  
jako dane 4  
jako metoda ogólna 65–71  
jako wynik 71–77  
jako wzorzec lokalnego rozwoju  
procesu obliczeniowego 30
- monitorowana 220
- nadawanie nazwy (za pomocą  
define) 12
- nazwa 12
- nierzetelna 391
- ogólna 164, 168  
selektor 175
- selektor ogólny 177
- parametry formalne 12

- procedura  
 pierwszorzędna w Lispie 75  
 porównanie z formą specjalną 392,  
 400  
 porównanie z funkcjami  
 matematycznymi 21–22  
 przekazująca wiele wartości 513  
 rekurencyjna  
 definicja 25  
 definiowanie bez użycia define  
 384  
 porównanie z procesem  
 rekurencyjnym 34  
 rzetelna 391  
 spamiętująca 266  
 treść 12  
 tworzenie  
 za pomocą define 12  
 za pomocą lambda-abstrakcji 61,  
 234, 235  
 wykonawcza  
 instrukcji 509  
 w evaluatorze  
 niedeterministycznym 418, 420  
 w evaluatorze z fazą analizy 385  
 w symulatorze maszyny  
 rejestrowej 514–521  
 zakres parametrów formalnych 28  
 złożona 11  
 użyta jak procedura pierwotna 13  
 proceduralna reprezentacja danych  
 90–93  
 dane modyfikowalne 254–255  
 procedure (niejawna forma begin  
 w ciele procedure) 217  
 procedure-body 369  
 procedure-environment 369  
 procedure-parameters 369  
 procedury pierwotne (oznaczone  
 przez *ns* nie wchodzą w skład  
 standardu IEEE języka Scheme)  
**\* 5**  
**+ 5**  
**- 5, 17**  
**/ 6**  
**< 17**  
**= 17**  
**> 17**  
**apply 181**
- procedury pierwotne  
 atan 172  
 car 85  
 cdr 85  
 cons 84  
 cos 68  
 display 86  
 eq? 142  
 error (*ns*) 67  
 eval (*ns*) 378  
 list 99  
 log 69  
 max 94  
 min 94  
 newline 86  
 not 19  
 null? 101  
 number? 145  
 pair? 109  
 quotient 325  
 random (*ns*) 51, 225  
 read 374  
 remainder 44  
 round 199  
 runtime (*ns*) 52  
 set-car! 247  
 set-cdr! 247  
 sin 68  
 symbol? 146  
 vector-ref 526  
 vector-set! 526  
 procedury wyższych rzędów 55  
 argumenty proceduralne 56–60  
 jako metody ogólne 65–71  
 jako wyniki 71–77  
 ścisła kontrola typów i - 344  
 w evaluatorze metacyklicznym 358  
 proces 1  
 iteracyjny 33  
 jako przetwarzanie strumieni  
 328–332  
 porównanie z procesem  
 rekurencyjnym 31–34, 238,  
 498, 585  
 realizacja w postaci maszyny  
 rejestrowej 498  
 kształt 33  
 liniowy iteracyjny 33  
 liniowy rekurencyjny 33

- proces  
  lokalny rozwój - 30  
  obliczeniowy 1  
  rekurencyjny 33  
  porównanie z procedurą  
    rekurencyjną 34  
  porównanie z procesem  
    iteracyjnym 31–34, 238, 498,  
    585  
  realizacja w postaci maszyny  
    rejestrowej 498–504  
rozgałęzający się rekurencyjnie  
  36–40  
wymagane zasoby 41  
zablokowany 305  
product 59  
  implementacja za pomocą  
    akumulacji 60  
product? 147  
programowanie  
  elementy 4  
  funkcyjne 225, 345–350  
    czas i - 347–350  
    współbieżność i - 349  
  imperatywne 230  
  niedeterministyczne, porównanie  
    z programowaniem w języku  
      Scheme 481  
sterowane danymi 169, 177–183  
  a analiza przypadków 357  
  w evaluatorze metacyklicznym  
    366  
  w interpreterze zapytań 461  
sterowane zapotrzebowaniami 317  
strukturnalne 377  
tragiczny styl 319  
w logice 429–433  
  historia 430, 432  
  język 431  
  komputery przeznaczone do - 432  
  porównanie z logiką matematyczną  
    454–459  
  reguła 433  
  w Japonii 432  
programy 1  
  jako dane 376–378  
  jako maszyny abstrakcyjne 376  
  komentarze 123
- programy  
  niedeterministyczne  
    analiza składniowa języka  
      naturalnego 412–416  
    łamigłówki logiczne 409–410  
  porównanie z programami  
    w języku Scheme 411, 412  
    trójkie pitagorejskie 408, 409  
  struktura 8–30  
  tworzenie przyrostowe 8  
  wynikowe 557  
  źródłowe 557  
projektowanie wielopoziomowe 139  
Prolog 407, 430  
promieniowanie kosmiczne 51  
prompt-for-input 375  
propagacja więzów 279–290  
propagate 275  
prostokąt, reprezentacja 90  
protokół zgodności pamięci  
  podręcznej 293  
przecinek, używany razem  
  ze wstecznym apostrofem 565  
przedział czasu, w kolejce zdarzeń  
  276  
przejście sygnału przez zero 338, 339  
przekazywanie  
  argumentów  
    obliczanych na żądanie 318, 393  
    przez nazwę 318, 393  
  komunikatów 92, 184–186  
    model środowiskowy i - 245  
    na przykładzie konta bankowego  
      219  
    w symulacji układów cyfrowych  
      272  
  wielu wartości 513  
przekierowanie  
  porównanie różnych stylów 185  
  wskaźnika 533  
  ze względu na typ 177  
przenoszenie języków 598  
przesyłanie komunikatów i rekursja  
  ogonowa 34  
przeszukiwanie  
  automatyczne 403  
    historia 407  
  systematyczne 407  
  w głąb 407

- przeszukiwanie  
     z nawrotami 407  
         chronologiczne 407  
         sterowane zależnościami 407  
 przetwarzanie  
     potokowe 291  
     sygnałów  
         model strumieniowy 336–339  
         przejście sygnału przez zero 338,  
             339  
         wyglądzanie funkcji 77  
         wyglądzanie sygnału 339  
 przewód, w układzie cyfrowym 267  
 przydzielanie pamięci  
     automatyczne 525  
     na stosie i rekursja ogonowa 577  
 przypisanie 214–231  
     błędy związane z - 229, 230  
     korzyści płynące z - 221–225  
     koszty wprowadzenia - 225–231  
 przystawanie modulo  $n$  50  
 pseudodzielenie wielomianów 211  
 pseudoreszta z dzielenia wielomianów  
     211  
 punkt  
     kontrolny w maszynie rejestrowej  
         524  
     reprezentowany jako para 89  
     stały 67–69  
         cosinusa 68  
         funkcji przekształconej 74  
         metodą kolejnych przybliżeń 77  
         obliczanie na kalkulatorze 68  
     pierwiastek czwartego stopnia jako  
         - 77  
     pierwiastek kwadratowy jako - 68,  
         72, 74  
     pierwiastek  $n$ -tego stopnia jako -  
         77  
     pierwiastek sześciennny jako - 72  
     unifikacja i - 470  
     w metodzie Newtona 72  
     złoty podział jako - 69  
 push 508  
 pusta lista 100  
 rozpoznawanie za pomocą null?  
     101  
 pusty strumień 312  
 put 179, 265
- qeval 453, 461  
 queens 124  
 query-driver-loop 460  
 quote (forma specjalna) 142  
     makrodefinicja read 477  
     read i - 374  
     quoted? 361  
 quotient (procedura pierwotna) 325
- Rabin Michael O. 54  
 rachunek  $\lambda$  (lambda) 62  
 Ramanujan Srinivasa 336  
 ramka  
     w interpreterze zapytań 445  
     reprezentacja 478  
 w języku graficznym 126, 133  
     odwzorowanie współrzędnych 133  
 w modelu środowiskowym 232  
     globalna 232  
     magazyn stanu lokalnego 239–243  
 rand 222  
     z ustawianiem 225  
 random (procedura pierwotna) 51  
     konieczność zastosowania  
         przypisania 216  
         MIT Scheme 225  
 random-in-range 225  
 random-numbers (strumień  
     niekończony) 346  
 rational (pakiet) 187  
 read (operacja maszyny rejestrowej)  
     490  
 read (procedura pierwotna) 374  
     jednoznakowe makrodefinicje 477  
     obsługa notacji kropki i ogona 467  
 read-eval-print-loop 551  
 real-part  
     dla danych ze znacznikami typu 175  
     programowanie sterowane danymi  
         181  
     reprezentacja biegunkowa liczb  
         zespolonych 173  
     reprezentacja prostokątna liczb  
         zespolonych 172  
 real-part-polar 175  
 real-part-rectangular 174  
 rear\_ptr 258  
 receive (procedura) 512  
 rectangular (pakiet) 179

- rectangular? 174  
Rees Jonathan A. 385  
reg (w maszynie rejestrowej) 489  
  symulacja 519  
register-exp 519  
register-exp-reg 519  
registers-modified 578  
registers-needed 578  
reguła  
  w języku zapytań 439–444  
  bez treści 440, 443, 464  
  stosowanie 451–452, 467–468, 482  
  w programowaniu w logice 433  
rejestr 483  
  docelowy 561  
  reprezentacja 507  
  śledzenie 524  
rekord w bazie danych 158  
rekurencja *zob.* rekursja  
rekurencyjny proces obliczeniowy 33  
  liniowy 33, 42  
  rozgałęziający się 36–40, 42  
rekursja 9  
  ogonowa 34  
  evlis 543  
  i odśmietanie 577  
  i przesyłanie komunikatów 34  
  w evaluatorze metacyklicznym 547  
  w evaluatorze z jawnie określonym sterowaniem 547–548, 554, 555  
  w języku Scheme 35  
  w kompilatorze języka Scheme 576  
  w środowiskowym modelu obliczeń 239  
operacje na drzewach 107  
sterowana danymi 204  
w maszynie rejestrowej 498–504  
w regułach 441  
wyrażanie skomplikowanych procesów 9  
relacje, obliczanie za pomocą - 430  
remainder (procedura pierwotna) 44  
remainder-terms 210  
remove 123  
remove-first-agenda-item! 274,  
  278  
reprezentacja danych, proceduralna 90–93  
require 405  
  forma specjalna 429  
rest-exp 363  
rest-operands 364  
rest-segments 277  
rest-terms 202, 206  
restore (instrukcja maszyny rejestrowej) 500, 520  
  implementacja 530  
  symulacja 518  
reszta modulo  $n$  50  
return (deskryptor łącznika) 561  
Reuter Andreas 308  
reverse 102  
  reguła 459  
  za pomocą składania 121  
rezolucja 430  
  klauzul hornowskich 430  
rezystancja  
  tolerancja rezistorów 93  
  wzór na rezystancję obwodu równoległego 93, 96  
right-branch 155, 164  
right-split 130  
Rivest Ronald L. 52, 156  
„robak” internetowy 598  
Rogers William Barton 126  
root rejestr 533  
rotate90 138  
round (procedura pierwotna) 199  
rozgałęzająca się rekursja 36–40  
rozgałęzające się rekurencyjnie  
  procesy obliczeniowe, złożoność 42  
równania rekurencyjne 2  
równanie różniczkowe 340  
  drugiego rzędu 342  
równość  
  liczb 17, 144, 527  
  list 143  
  niewrażliwość na odniesienia a - 228  
  symboli 142  
  w ogólnym systemie arytmetycznym 191  
różniczkowanie  
  numeryczne 73  
reguły 149

- różniczkowanie  
     symboliczne 144–149, 182  
     zasady 144  
 Runkle John Daniel 127  
 runtime (procedura pierwotna) **52**  
 rysowniki 126  
     operacje 127  
         wyższych rzędów 132  
     przekształcanie i łączenie 137  
     reprezentacja proceduralna 135  
 rząd wielkości 41–42  
     notacja 42  
 rzutowanie typów 192–199  
     procedura 192  
     tablica 192  
     w arytmetyce wielomianów 204  
     w systemach operacji algebraicznych 208
- same** (reguła) **440**  
**same-variable?** **146**, 201  
 samoobliczające się wyrażenie 356  
**save** (instrukcja maszyny rejestrowej)  
     500, 520  
     implementacja 530  
     symulacja 518  
 scalanie w trakcie odśmiecania 532  
**scale-list** **104**, **105**, **401**  
**scale-stream** **323**  
**scale-tree** **111**  
**scale-vect** 135  
**scan** (rejestr) 533  
**scan-out-defines** 381  
**Scheme** 3  
     evaluator, implementacja w postaci  
         układu scalonego 538  
     historia 3  
     interpretator napisany w C 601  
**scheme-number** (pakiet) 187  
**scheme-number->complex** **192**  
**scheme-number->scheme-number**  
     **197**  
**search** **66**  
**segment-queue** **277**  
**segment-time** **277**  
**segments** **277**  
**segments->painter** **135**  
 sekcja krytyczna 297–301  
     implementacja 304–307
- sekcja krytyczna  
     wiele współdzielonych zasobów  
         301–304  
 sekwencja wyrażeń  
     w następcu wyrażenia  
         warunkowego cond 18  
     w treści procedury 12  
 selektor 83  
     jako bariera abstrakcji 88  
     ogólny 175, 177  
**self-evaluating?** **361**  
**semafor** 304  
     ogólny 304  
     rozmiaru *n* 307  
     operacja P 304  
     operacja V 304  
**sequence->exp** **363**  
**serialized-exchange** **302**  
     z unikaniem zakleszczenia **308**  
**set!** (forma specjalna) **216**  
     w modelu środowiskowym 236  
     wartość 216  
**set-car!** (procedura pierwotna) **247**  
     implementacja proceduralna **255**  
     implementacja wektorowa 529  
     wynik 247  
**set-cdr!** (procedura pierwotna) **247**  
     implementacja proceduralna **255**  
     implementacja wektorowa 529  
     wynik 247  
**set-contents!** **508**  
**set-current-time!** **277**  
**set-front-pointer!** **258**  
**set-instruction-execution-proc!**  
     **514**  
**set-rear-pointer!** **258**  
**set-register-contents!** **505**, **511**  
**set-segments!** **277**  
**set-signal!** **270**, **273**  
**set-value!** **283**, **288**  
**set-variable-value!** **369**, **371**  
**setup-environment** **373**  
 Shamir Adi 52  
**shrink-to-upper-right** **137**  
 sieć wiązów 280  
**signal-error** **552**  
 silnia 31  
     bez letrec i define 384  
     strumień nieskończony 324

- simple-query** **462**  
Simpsona metoda całkowania numerycznego **59**  
**sin** (procedura pierwotna) **68**  
**singleton-stream** **475**  
sinus  
  granica szeregu potęgowego **326**  
  przybliżenie dla małych kątów **43**  
sito Eratostenesa **320**  
**sieve** **321**  
**SKETCHPAD** **279**  
składanie wyrażeń **5**  
składnia  
  abstrakcyjna  
    w evaluatorze metacyklicznym **356**  
    w interpreterze zapytań **460**  
  języka programowania **11**  
  wyrażeń, opis **12**  
skracanie  
  funkcji wymiernych **211–212**  
  ułamków **87, 88**  
słowo kluczowe **589, 594**  
**smallest-divisor** **49**  
  wersja bardziej efektywna **53**  
**Smalltalk** **280**  
**Solomonoff Ray** **222**  
**solve** (rozwiązywanie równań różniczkowych) **340, 341**  
  z leniwymi listami **402**  
  z wyłuskiwaniem definicji **382**  
spamiętywanie **40, 266**  
  **delay** **317**  
  obliczanie na żądanie i - **325**  
  wartości domknięć **393**  
specyfikacje danych, algebraiczne **91**  
**split** **133**  
sprzężenie zwrotne, modelowanie za pomocą strumieni **339**  
**sqrt** **23**  
  granica strumienia **332**  
  jako punkt stały **69, 72, 74, 75**  
  metodą kolejnych przybliżeń **77**  
  metodą Newtona **74, 75**  
  obliczane przez maszynę rejestrową **494**  
  struktura blokowa **29**  
  w modelu środowiskowym **243–245**  
**sqrt-stream** **328**  
**square** **11**  
  w środowiskowym modelu obliczeń **234–235**  
**square-limit** **132, 133**  
**square-of-four** **132**  
**squarer** (więzy) **288**  
**squash-inwards** **138**  
**stack-inst-reg-name** **518**  
Stallman Richard M. **280, 407**  
stała, definicja w maszynie rejestrowej **505**  
stan  
  lokalny **214–231**  
  przechowywany w ramkach **239–243**  
  znika w sformułowaniu strumieniowym **348**  
**start** maszyny rejestrowej **506, 511**  
**start-eceval** **595**  
**start-segment** **89, 136**  
**statements** **578**  
Steele Guy Lewis Jr. **3, 34, 231, 280**  
sterownik maszyny rejestrowej **484–487**  
diagram **486**  
**stos** **34**  
  implementacja rekursji w maszynie rejestrowej **498–504**  
  ramki **541**  
  reprezentacja **508, 530**  
stosowana kolejność obliczania  
  porównanie z normalną kolejnością obliczania **20, 48, 390–392**  
  w Lispie **16**  
stosowanie procedur w modelu środowiskowym **236–239**  
Stoy Joseph E. **15, 46**  
Strachey Christopher **75**  
**stream-append** **333**  
**stream-append-delayed** **474**  
**stream-car** **312, 314**  
**stream-cdr** **312, 314**  
**stream-enumerate-interval** **315**  
**stream-filter** **315**  
**stream-flatmap** **475, 479**  
**stream-for-each** **313**  
**stream-limit** **331**  
**stream-map** **313**  
  wersja wieloargumentowa **318**

- stream-null?** 312  
 w MIT Scheme 312  
**stream-ref** 313  
**stream-withdraw** 348  
**struktura**  
 blokowa 29–30, 379–381  
 w języku zapytań 482  
 w modelu środowiskowym 243–246  
**danych, hierarchiczna** 97  
**listowa** 85  
 modyfikowalna 247–251  
 pamięć 524–537  
 reprezentacja przy użyciu wektorów 526–531  
**sterująca** 454  
**strumień** 214, 309–350  
 implementacja za pomocą list odroczonych 311–314  
 nieskończony 319–327  
 liczb losowych 346  
 modelujący sygnał 336–339  
**par** 332–336  
 reprezentujący szereg potęgowy 326  
 scalanie 325, 333, 335, 349  
 scalanie jako relacja 350  
**silni** 324  
 sum częściowych szeregu 328  
 obliczenia odroczone i - 339–344  
**pusty** 312  
 uwikłana definicja 321–324  
 zaimplementowany jako lista leniwa 400–403  
 zastosowanie w interpreterze ramek 446  
 zastosowanie w interpreterze zapytań 453  
**styl imperatywny a zorientowany na wyrażenia** 290  
**styl zorientowany na wyrażenia a imperatywny** 290  
**sub** (procedura ogólna) 186  
**sub-complex** 171  
**sub-interval** 94  
**sub-rat** 84  
**sub-vect** 135  
**subsets** (podzbiory danego zbioru) 112  
**sum** 57  
 wersja iteracyjna 59  
 za pomocą akumulacji 60  
**sum-cubes** 56  
 procedura wyższego rzędu 57  
**sum-integers** 56  
 z procedurami wyższych rzędów 57  
**sum-odd-squares** 113, 116  
**sum-of-squares** 12  
 w modelu środowiskowym 236–238  
**sum-primes** 311  
**sum?** 147  
**suma**  
 ciągu elementów 57  
 częściowa szeregu, strumień 328  
 logiczna 268  
**sumator**  
 kaskadowy 272  
 pełny 269  
 Sussman Gerald Jay 3, 34, 280, 407  
 Sutherland Ivan 279  
 sygnał cyfrowy 267  
**symbol**  
 katalogowanie 528  
 niepowtarzalność 252  
 reprezentacja 528  
**symbol-leaf** 163  
**symbol?** (procedura pierwotna) 146  
 implementacja przy użyciu wskaźników ze znacznikiem typu 530  
 typy danych 190  
**symbole** 140  
 cytowanie 142  
 równość 142  
 symboliczne różniczkowanie 144–149, 182  
**symbols** 164  
**symulacja**  
 jako narzędzie projektowania maszyn 553  
 monitorowanie wydajności maszyny rejestrowej 522  
 sterowana zdarzeniami 267  
 układów cyfrowych 267–279  
 bramki elementarne 270–272  
 implementacja kolejki zdarzeń 276–279  
 kolejka zdarzeń 274–275

- symulacja  
układów cyfrowych  
przykładowa symulacja 275–276  
reprezentacja przewodów 272–274
- symulator maszyny rejestrowej  
505–524
- SYNC 309
- system THE 304
- szachy, problem ośmiu hetmanów  
124, 412
- szereg 57
- nieokończony 470
- potęgowy jako strumień 326
- całkowanie 326
- dodawanie 326
- dzielenie 327
- mnożenie 326
- suma częściowa, strumień 328
- sumowanie, akcelerator 329
- szeregowanie 297–301
- szerokość przedziału 94
- ścieżki danych maszyny rejestrowej  
484–487
- diagram 485
- śledzenie
- instrukcji w maszynie rejestrowej  
    523
- przypisań w maszynie rejestrowej  
    524
- średnik 11
- oznaczenie komentarza 123
- środki
- abstrakcji 4
- define 8
- łączenia 4
- środowisko 8, 232
- globalne 8, 233
- w metacyklicznym evaluatorze  
    języka Scheme 373
- jako kontekst przy obliczaniu  
    wartości 10
- kompilacji 592
- otwarte kodowanie i - 594
- otaczające 232
- przemianowanie a - 481
- w interpreterze zapytań 482
- wiązanie składniowe i - 29
- wykonania programu 593
- środowiskowy model obliczeń 214,  
232–246
- definicje wewnętrzne 243–246
- evaluator metacykliczny i - 354
- przekazywanie komunikatów 245
- przykład stosowania procedur  
236–239
- reguły obliczania 233–236
- rekursja ogonowa 239
- stan lokalny 239–243
- struktura środowiska 233
- #t 17**
- tableau 330
- tablica 261–267
- dwuwymiarowa 263–264
- jednowymiarowa 261–262
- lokalna 264–265
- n-wymiarowa 266
- operacji i typów 179
- implementacja 265
- konieczność zastosowania  
  przypisania 216
- porównywanie kluczy 265
- programowanie sterowane danymi  
178
- rejestrów
- w symulatorze maszyny  
    rejestrowej 509
- reprezentowana jako drzewo binarne  
  i lista nieuporządkowana 266
- rzutowania typów 192
- spamiętywanie obliczonych wartości  
266
- szkielet 261
- użyta w kolejce zdarzeń symulacji  
277
- tablicowanie 40, 266
- tack-on-instruction-sequence 581**
- tagged-list? 361**
- tangens
- granica szeregu potęgowego 327
- jako ułamek łańcuchowy 71
- Teitelman Warren 3
- teoria
- funkcji rekurencyjnych 378
- liczb 49
- term-list 201**

- test** (instrukcja maszyny rejestrowej)  
488  
symulacja 516  
test na bycie zerem (ogólny) 191  
dla wielomianów 206  
test pierwszości Fermata 49–52  
variant 54  
**test-and-set!** 306  
**test-condition** 517  
**text-of-quotation** 361  
teza Churcha-Turinga 377  
Thatcher James W. 91  
**the-cars**  
rejestr 528, 533  
wektor 526  
**the-cdrs**  
rejestr 528, 533  
wektor 526  
**the-empty-stream** 312  
w MIT Scheme 312  
**the-empty-termlist** 202, 206  
**the-global-environment** 373, 551  
 $\theta(f(n))$  (theta od  $f(n)$ ) 42  
**thunk**, pochodzenie terminu 392  
**timed-prime-test** 52  
TK, Solver 280  
tłumienie przez uśrednienie 69  
tożsamość i zmiana  
współdzielenie danych i - 251  
znaczenie 227–230  
**transform-painter** 137  
**transpose** (transpozycja macierzy)  
120  
**tree->list...** 157  
**tree-map** 112  
treść procedury 12  
trójkąt Pascala 41  
trójkąt pitagorejskie  
wyznaczane przy użyciu programu  
niedeterministycznego 408, 409  
wyznaczane przy użyciu strumieni  
335  
**true** 17  
**true?** 368  
**try-again** 408  
Turing Alan M. 376, 379  
Turner David 121, 334  
twierdzenie Lamégo 48  
tworzenie programów, przyrostowe 8
- typ**  
przekierowanie ze względu na - 177  
wskaźnika 526  
**type-tag** 174  
z użyciem typów danych języka  
Scheme 190  
**typy**  
danych  
w językach ze ścisłą kontrolą  
typów 344  
w Lispie 190  
hierarchia 194–199  
hierarchia w algebrze symbolicznej  
207–208  
nadtyp 194  
operacje na elementach różnych  
typów 191  
podnoszenie 195  
podtyp 194  
rzutowanie w dół 195  
rzutowanie w górę 198  
wiele nadtypów i podtypów 196  
wieża - 195  
wyprowadzanie 345
- układ**  
całkujący, sygnały 336  
scalony, implementujący evaluator  
języka Scheme 538  
ułamek łańcuchowy 70  
*e* jako - 70  
tangens jako - 71  
złoty podział jako - 70  
**unev** (rejestr) 539  
unifikacja 450–451  
implementacja 468–471  
porównanie z dopasowywaniem  
wzorców 451, 453  
wynalezienie algorytmu 430  
**unify-match** 469  
**union-set** 150  
reprezentacja zbiorów za pomocą  
drzew binarnych 158  
list nieuporządkowanych 152  
list uporządkowanych 154  
**unique** (w języku zapytań) 480  
**unique-pairs** 124  
UNIX 556, 598  
**unknown-expression-type** 552

- unknown-procedure-type **552**  
up-split **132**  
update-insts! **513**  
upper-bound **94**  
upraszczanie wyrażeń algebraicznych  
    148–149  
user-initial-environment (MIT  
    Scheme) **378**  
user-print **375**  
    zmodyfikowane pod kątem  
        skompilowanego kodu **596**  
utożsamianie nazw **229**  
utrzymywanie wiarygodności **407**  
używane notacje  
    odpowiedź interpretera **5**  
składnia wyrażeń **12**
- val (rejestr) **539**  
value-proc **516**  
variable **201**  
variable? **146, 361**  
vector-ref (procedura pierwotna)  
    **526**  
vector-set! (procedura pierwotna)  
    **526**  
verbs **412**
- Wagner Eric G. **91**  
Walker Francis Amasa **127**  
Wallis John **59**  
Wand Mitchell **353**  
wartość **7**  
    bezwzględna **16**  
    kombinacji **6**  
    nieokreślona  
        define **7**  
        display **86**  
        if bez alternatywy **278**  
        newline **86**  
        set! **216**  
        set-car! **247**  
        set-cdr! **247**  
    wyrażenia **7**  
warunek, sprawdzanie w maszynie  
    rejestrowej **486**  
Waters Richard C. **118**  
wcięcia **7**  
weight **164**  
weight-leaf **163**
- wektor (matematyczny)  
    operacje **119, 135**  
ramka (w języku graficznym) **133**  
reprezentowany jako ciąg **119**  
reprezentowany jako para **135**  
wektor (struktura danych) **525**  
Weyl Hermann **79**  
węzeł drzewa **9**  
wheel (reguła) **440, 458**  
wiązanie **27**  
    głębokie **372**  
model środowiskowy **232**  
składniowe **29**  
    struktura środowiska i - **591**  
width **95**  
wiedza deklaratywna i imperatywna  
    21  
obliczenia niedeterministyczne i -  
    **404**  
porównanie **429**  
    programowanie w logice **430–432,**  
        **454**  
wielkość niezmiennicza procesu  
    iteracyjnego **45**  
wielomian  
    wyrazy **201**  
    zmienna nieoznaczona **200**  
wielomiany  
    arytmetyka - **200–212**  
    gęste **205**  
    hierarchia typów **207–208**  
    postać kanoniczna **208**  
    rzadkie **205**  
    schemat Hornera **118**  
wieża typów **195**  
więzy  
    pierwotne **280**  
    propagacja **279–290**  
Wilde Oscar (w parafrazie Perlisa) **7**  
Wiles Andrew **50**  
Winograd Terry **407**  
Winston Patrick Henry **408, 417**  
Wisdom Jack **3**  
Wise David S. **318**  
withdraw **216**  
without-interrupts **306**  
własność domknięcia **82**  
    brak w wielu językach **98**  
    cons **97**

- własność domknięcia
  - operacji języka graficznego 126, 128
  - w algebrze abstrakcyjnej 97
- wprowadzanie wyrażeń 7
- Wright E. M. 324
- Wright Jesse B. 91
- wskaźnik
  - w notacji pudełkowo-wskaźnikowej 97
  - ze znacznikiem typu 526
- współbieżność 290–309
  - mechanizmy kontroli 296–309
  - poprawność programów współbieżnych 294–296
  - programowanie funkcyjne i - 349
  - zakleszczenie 307–308
- współczynniki dwumianowe 41
- współdzielenie danych 251–254, 293
- wybór niedeterministyczny 406
- wydajność kompilacji 559
- wydawanie reszty 38–40, 102
- wyglądzanie
  - funkcji 77
  - sygnału 339
- wykluczanie wzajemne 304
- wykonanie instrukcji
  - w symulatorze maszyn rejestrowych 509
- wykrzyknik w nazwach 216
- wyłuskiwanie definicji wewnętrznych 380
  - w kompilatorze 592, 593
- wymuszenie obliczenia domknięcia 393
- wypisywanie, operacje pierwotne 86
- wyprowadzanie typów 345
- wyrazy wielomianu 201
- wyrażenie
  - algebraiczne 199
  - reprezentacja 146–149
  - różniczkowanie 144–149
  - upraszczanie 148–149
- pierwotne 4
  - liczby 5
  - nazwy procedur pierwotnych 5
  - nazwy zmiennych 7
  - obliczanie wartości 9
- wyrażenie
  - pochodne, w ewaluatorze 364–365
  - dodawanie do ewaluatora z jawnie określonym sterowaniem 550
  - samoobliczające się 356
  - symboliczne 82
  - warunkowe
    - cond 16
    - if 18
  - złożone 5
    - jako operator kombinacji 20
- wyszukiwanie
  - w drzewie binarnym 154
  - wywołania zagnieżdżone car i cdr 100
  - wzorzec 435–437
  - wzorzec (proste zapytanie)
    - pasujące podstawienia zmiennych 437
  - wzór interpolacyjny Lagrange'a 200
  - wzrost
    - liniowy 33
    - wykładniczy
  - liczb Fibonacciego 36
- x-point 89
- xcor-vect 135
- Xerox Palo Alto Research Center 3, 280
- y-point 89
- ycor-vect 135
- Yochelson Jerome C. 532
- Zabih Ramin 408
- zagłodzenie 305
- zakleszczenie 307–308
  - likwidacja 308
  - unikanie 307
- zakres zmiennej 28
  - definicje wewnętrzne 379
  - parametrów formalnych procedury 28
  - w wyrażeniu let 64
- założenie o domknięciu świata 457
- zaokrąglenie, błąd 5
- zapytanie 432
  - proste 435–437
  - podstawienia spełniające - 435–437

- zapytanie  
proste  
  przetwarzanie 446, 447, 452–453,  
    462  
złożone 437–439  
  podstawienia spełniające - 438–439  
  przetwarzanie 447–449, 462–464,  
    480, 481
- zasada  
  minimalnych zobowiązań 173  
  rezolucji 430  
  ukrywania informacji 217
- zasłanianie  
  wiązań 232  
  zmiennych wolnych 28
- zasoby współdzielone 301–304
- zastosowanie procedury, oznaczenie  
  kombinacji 6
- zbiór 150  
  baza danych jako - 158  
  operacje na - 150  
  permutacje 123  
  podzbiory 112  
  reprezentowany jako drzewo binarne  
    154–158  
  reprezentowany jako lista  
    nieuporządkowana 150–152  
  reprezentowany jako lista  
    uporządkowana 152–154
- Zetalisp 3
- Zilles Stephen N. 91
- Zippel Richard E. 212
- zliczanie instrukcji w maszynie  
  rejestrowej 523
- złamane serce 533
- złącze  
  w sieci więzów 280
- w systemie więzów  
    operacje 283  
      reprezentacja 286
- złoty podział 37
- jako punkt stały 69
- jako ułamek łańcuchowy 70
- złożenie funkcji 76
- złożoność  
  liniowa 42  
  liniowych iteracyjnych procesów  
    obliczeniowych 42
- liniowych rekurencyjnych procesów  
    obliczeniowych 42
- logarytmiczna 42, 44, 154
- rozgałęziających się rekurencyjnie  
    procesów obliczeniowych 42
- wykładnicza 42
- zmiana i tożsamość  
  współdzielenie danych i - 251
- znaczenie 227–230
- zmienna 7  
  lokalna 62–65  
  nieoznaczona wielomianu 200  
  stanu 33, 214  
    lokalna 215–221
- wartość 232
- we wzorcu 435  
  reprezentacja 461, 476–478
- wolna 28, 232
- zakres 28
- związana 27
- znacznik  
  końca listy 99  
  typu 168, 173, 526  
    dwupozymowy 189  
    wskaźnika 526
- znak  
  systemu 374  
    ewaluatora metacyklicznego 375  
    ewaluatora niedeterministycznego  
      427
- interpretora zapytań 460  
  leniwego ewaluatora 395
- zachęty 374  
  ewaluatora metacyklicznego 375  
  ewaluatora niedeterministycznego  
      427
- interpretora zapytań 460  
  leniwego ewaluatora 395
- zapytania, w nazwach predykatów  
  23