Przegląd języków i paradygmatów programowania

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Przegląd języków i paradygmatów programowania

Jarosław Bylina
Beata Bylina

Lublin 2011

Instytut Informatyki UMCS
Lublin 2011

Jarosław Bylina (Instytut Matematyki UMCS)
Beata Bylina (Instytut Matematyki UMCS)

Przegląd języków i paradygmatów programowania

Recenzent: Joanna Domańska

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-00-8

Spis treści

Wstęp

Niniejszy skrypt powstał na bazie doświadczeń autorów w nauczaniu przedmiotów informatycznych na Wydziale Matematyki, Fizyki i Informatyki UMCS, w tym (i przede wszystkim) przedmiotu „Języki i paradygmaty programowania". Ma też za zadanie służyć jako pomoc w przyswojeniu tego przedmiotu na kierunku „Informatyka". Jednakże, ponieważ prezentuje przegląd wielu zagadnień związanych z programowaniem i różnymi językami programowania, to może być również przydatny dla studentów innych kierunków, zainteresowanych tematyką skryptu, a także dla absolwentów, którzy chcieliby uzupełnić wiedzę nabytą na studiach jakiś czas temu.

Z drugiej strony postaramy się nie powtarzać materiału, który omawiają inne przedmioty (jak na przykład „Programowanie obiektowe").

Celem niniejszego opracowania jest pokazanie różnorodności istniejących języków programowania oraz podejść do programowania, a także istotnych cech i różnic języków programowania. Chcemy także uzmysłowić czytelnikowi pewne aspekty implementacji niektórych elementów języków, które na co dzień ukryte są przed użytkownikiem języka (czyli programistą), ale powinny mieć wpływ na wybór języka oraz wybór narzędzi dostępnych w owym języku przez programistę.

Jasne jest bowiem, że żaden język programowania (ani też żaden paradygmat) nie nadaje się do wszelkich zastosowań. Programista (czy też jego szef — analityk, projektant itp.) musi więc wybierać język najlepiej przystosowany do danego zadania, biorąc pod uwagę nie tylko modę czy znajomość danego języka w zespole[1], ale przede wszystkim dostosowanie języka[2] do zadania programistycznego.

Co znaczy — użyte w tytule skryptu — słowo 'paradygmat'? Pochodzi z języka greckiego, w którym słowo $\pi\alpha\rho\acute{\alpha}\delta\epsilon\iota\gamma\mu\alpha$ (*parádeigma*) oznacza wzorzec lub przykład. Internetowy słownik języka polskiego [83] definiuje

[1] Co, naszym zdaniem, jest sprawą drugo- lub nawet trzeciorzędną, bo sztuka programowania nie jest znajomością konkretnego języka, lecz znajomością technik i metod projektowania algorytmów. Kodowanie w konkretnym języku jest dopiero kolejnym (dość banalnym) krokiem, a dobry programista nauczy się nowego języka w ciągu kilku dni.

[2] W tym także jakości i dostępności jego otoczenia: kompilatorów, bibliotek, dokumentacji...

paradygmat między innymi jako ‘przyjęty sposób widzenia rzeczywistości w danej dziedzinie’.

W wyrażeniu ‘paradygmaty programowania’ nie chodzi o wzorcowy sposób programowania czy też o przykłady poprawnych programów. Mówiąc o paradygmatach programowania, mamy na myśli raczej zestaw typowych dla danej grupy języków mechanizmów udostępnionych programiście oraz zbiór sposobów interpretacji tych mechanizmów przez semantykę języka. Czyli — jak rzeczywistość (tak zewnętrzna, opisywanego świata, jak i wewnętrzna, maszynowa) jest postrzegana przez pryzmat danego języka.

Tak też będziemy rozumieć wyrażenie ‘paradygmaty programowania’. Nie jest jednak naszym celem ścisłe i naukowe segregowanie i opisywanie poszczególnych paradygmatów (jak to jest na przykład w [51]), lecz raczej chodzi o podejście praktyczne, bliskie programiście. Chcemy omówić ogólny podział paradygmatów programowania, biorąc pod uwagę główne paradygmaty i przykłady języków je reprezentujących (Rozdział 1). Następnie zajmiemy się wybranymi problemami opisu języków programowania (Rozdział 2), po czym przejdziemy do ogólnych zagadnień dotyczących mechanizmów powszechnie występujących w programowaniu, jak dane i ich typy (Rozdział 3) oraz podprogramy i obiekty (Rozdział 4). Poświęcimy też dwa rozdziały specyficznym paradygmatom: funkcyjnemu (z językiem Haskell, Rozdział 5) i logicznemu (z językiem Prolog, Rozdział 6). W końcu skupimy się na języku Python (Rozdział 7) łączącym wiele paradygmatów. Całość zakończymy rozważaniami na temat mniej rozpowszechnionych paradygmatów programowania, ale zyskujących coraz większą popularność — współbieżnego, równoległego i rozproszonego (Rozdział 8).

Wszystkich zachęcamy także, do zajrzenia na stronę www:
`http://kokos.umcs.lublin.pl/ksiazka-pjipp`
gdzie można (po rejestracji i zalogowaniu się) znaleźć materiały uzupełniające do skryptu, erratę (bo błędów nie unikniemy na pewno) oraz forum dla dociekliwych czytelników. Na tym forum będziemy też omawiali — jeśli zajdzie taka potrzeba — rozwiązania zadań i odpowiedzi na pytania, które będzie można znaleźć na zakończenie każdego rozdziału. Szczególnie dotyczy to zadań oznaczonych gwiazdką (*), czyli trudniejszych.

Książka niniejsza powstała w oparciu o wiele prac (pełna bibliografia znajduje się na stronie 205), ale najważniejsze to [2, 9, 37, 54, 68].

Cokolwiek jest przydatnego w tej książce, zawdzięczamy to ludziom, którzy przez długie lata byli — i nadal są — naszymi informatycznymi mentorami. Są to: doc. dr Światomir Ząbek, dr hab. Przemysław Stpiczyński, dr Jerzy Mycka. Za wszelkie uwagi dziękujemy też pierwszemu czytelnikowi tej książki — Jerzemu Bylinie[3].

[3] To mój Tata! — *przypis JB*

Rozdział 1

Podział paradygmatów programowania

1.1. Podział podstawowy

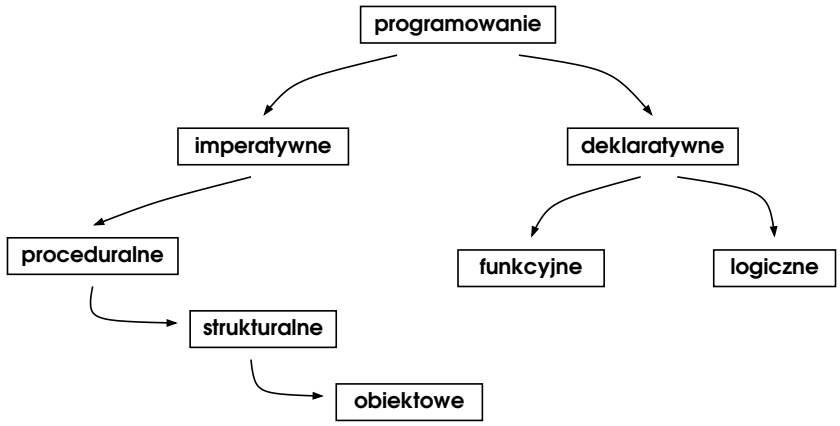
Wszystkie języki programowania można podzielić na dwie główne grupy[1]:

— *imperatywne*,
— *deklaratywne*.

Języki imperatywne charakteryzują się tym, że programista wyraża w nich czynności (w postaci rozkazów), które komputer ma w pewnej kolejności wykonywać. Program jest więc listą rozkazów do wykonania, stąd nazwa *imperatywne* („rozkazowe").

W językach deklaratywnych jest inaczej. Tutaj programista pisząc program podaje (*deklaruje*) komputerowi pewne zależności oraz cele, które program ma osiągnąć. Nie podaje się jednak wprost sposobu osiągnięcia wyników.

Innymi słowy, języki imperatywne mówią komputerowi, **jak** ma osiągnąć wynik (choć samego wyniku nie określają wprost), natomiast języki deklaratywne opisują, **co** ma być osiągnięte (choć nie podają na to bezpośredniego sposobu). Ten podział przedstawia Rysunek 1.1.



Rysunek 1.1. Podział głównych paradygmatów programowania

1.2. Główne paradygmaty: programowanie imperatywne a deklaratywne

Na Rysunku 1.1 widać także dalszy, drobniejszy, podział wspomnianych wyżej dwóch głównych paradygmatów programowania.

[1] Choć, prawdę mówiąc, całkiem sporo języków łączy elementy z obu tych grup.

W programowaniu imperatywnym — jak wyżej to wspomnieliśmy — program jest po prostu listą instrukcji (mniej lub bardziej elementarnych), które mają być wykonywane kolejno. Jednakże, siłą maszyn liczących[2] jest umiejętność wielokrotnego powtarzania pewnych czynności zapisanych raz — czyli wykonywania pętli.

Żeby było to możliwe, musimy mieć do dyspozycji rozkaz „zaburzający" zapisaną sekwencję poleceń, a podstawową i pierwotnie stosowaną, pełniącą tę funkcję instrukcją jest *rozkaz!skoku*. Zestaw instrukcji zawierający elementarne rozkazy operacji na pewnych danych wraz z rozkazami skoków bezwarunkowych i warunkowych stanowi trzon każdego z kodów maszynowych. Jest to też najbardziej naturalny i pierwotny język (czy też rodzina języków: kody maszynowe) dla wszelkiego rodzaju komputerów (i podobnych maszyn), bowiem są one budowane począwszy od ich powstania do dziś na bazie abstrakcyjnej *architektury von Neumanna* (lub architektur blisko pokrewnych).

Architektura von Neumanna [78] zakłada w uproszczeniu, że:

— maszyna składa się z pamięci oraz jednostki centralnej, która wykonuje rozkazy (procesora);
— rozkazy oraz dane zapisane są w tej samej pamięci w ten sam sposób;
— rozkazy są kolejno z pamięci wczytywane do jednostki centralnej i wykonywane;
— każdy rozkaz powoduję zmianę *stanu maszyny* rozumianego jako zawartość całej pamięci włącznie z rejestrami i znacznikami procesora; rozkazy mogą więc zmieniać wewnętrzne ustawienie jednostki centralnej, w tym miejsce, z którego będzie czytany następny rozkaz.

W praktyce komputery budowane są (i działają) właśnie tak (choć z pewnymi drobnymi modyfikacjami). W związku z tym trzeba wyraźnie powiedzieć, że **najbardziej naturalnym paradygmatem dla maszyny jest paradygmat imperatywny**. Innymi słowy: maszyna musi dostać kolejne kroki do wykonania, żeby mogła coś zrobić. Z drugiej strony, **dla człowieka dużo wygodniejszym sposobem komunikowania poleceń jest podanie, co ma być osiągnięte, bez wdawania się w szczegóły wykonania — a więc paradygmat deklaratywny**. Ta dwoistość jest między innymi powodem istnienia tak szerokiego wachlarza języków programowania realizujących różnorakie paradygmaty. Widoczne jest to szczególnie w nowoczesnych językach programowania łączących różne paradygmaty, ale od zarania informatyki — na tyle na ile sprzęt pozwalał — widać usilną chęć połączenia obu tych często przeciwstawnych tendencji.

[2] Zauważyła to już w swoich notatkach w XIX wieku Ada Lovelace (z domu Byron) [70] — której imieniem nazwano zresztą jeden z języków programowania.

1.2.1. Programowanie proceduralne

Od dawna (właściwie od samego początku) maszyny będące w powszechnym użyciu dostarczają zwykle mechanizmu *stosu*[3], który — wraz z rozkazami do jego obsługi — umożliwia *programowanie proceduralne* będące podparadygmatem programowania imperatywnego. W programowaniu proceduralnym mamy wydzielone elementy kodu zwane *podprogramami* (w różnych językach i kontekstach mówi się tu o procedurach, funkcjach, metodach, operacjach...), które mogą być wielokrotnie — także *rekurencyjnie* — wywoływane z różnymi *parametrami*.

W kodzie maszynowym zwykle nie ma ograniczenia na liczbę *punktów wejścia podprogramu* (czyli miejsc, od których dany podprogram może zostać rozpoczęty)[4] ani na liczbę jego *punktów wyjścia* (czyli miejsc, w których dany podprogram może się zakończyć)[5]. Jednakże, języki wyższego poziomu ograniczają zwykle (właściwie zawsze) liczbę punktów wejścia do jednego[6].

Warto zauważyć, że programowanie proceduralne umożliwiło powstanie techniki (czy też metodologii) programowania *bottom-up* [68]. Polega ona na projektowaniu oprogramowania od małych części, podalgorytmów, które zapisywane są w postaci podprogramów. Z tych małych cegiełek budowane są większe podprogramy, z nich jeszcze większe, i tak dalej, aż do powstania całego, zamierzonego dzieła programistycznego. Takie podejście ułatwia

[3] Stos, pod względem możliwości obliczeniowych, nie rozszerza architektury von Neumanna (co więcej: jest zbudowany w jej ramach), ale ułatwia pewne czynności. Klasyczny stos to struktura danych, która pozwala na następujące operacje:
— `top`: sprawdzenie, co jest na szczycie stosu;
— `push`: włożenie danej na szczyt stosu (włożona dana staje się nowym szczytem stosu);
— `pop`: zdjęcie danej ze szczytu stosu.

Na stos można wkładać dowolnie wiele danych (oczywiście ograniczone jest to pamięcią przeznaczoną na stos), a ich zdejmowanie odbywa się w kolejności odwrotnej, co jest ze wszech miar wygodne (jak się to okaże na przykład w Rozdziale 4). Tak więc, jeśli:
— na pusty stos włożymy kolejno elementy a, b, c;
— potem zdejmiemy jeden (c);
— włożymy d;
— zdejmiemy dwa (będą to kolejno: d, b);
— włożymy e, f;
to w tym momencie na stosie są (patrząc od strony szczytu stosu) elementy: f, e, a.

Stos maszynowy zwykle pozwala na sprawdzenie zawartości dowolnego jego miejsca względem szczytu (lecz niekoniecznie względem dna!), nie tylko samego szczytu.

[4] Bo start podprogramu regulowany jest adresem, od którego należy rozpocząć jego wykonanie, a rozkaz `!CALL` (czy też podobny) można wykonać z dowolnym adresem, także w środku podprogramu.

[5] Bo rozkaz powrotu z podprogramu `RET` (czy też podobny) może znaleźć się w wielu miejscach, także na przykład jednocześnie w kilku gałęziach selekcji.

[6] Jeśli chodzi o punkty wyjścia, to także zdarzają się takie ograniczenia, choć zdecydowanie rzadziej. Przykładem może być tradycyjna specyfikacja języka Pascal [65]. Chociaż i tam jest — zdecydowanie odradzana, ale istniejąca — instrukcja `goto`.

znacznie pracę nad programem, a także myślenie o nim, projektowanie go i przede wszystkim likwidowanie błędów. Ta technika pozwoliła na rozwój dwóch ważnych aspektów programowania:

— programowania zespołowego — każdy z programistów dostaje do wykonania swoje podzadanie składające się z wydzielonych (i ściśle wyspecyfikowanych przez kierującego zespołem) podprogramów; z owych podprogramów budowane jest całe oprogramowanie;
— bibliotek oprogramowania — takie biblioteki są przecież zbiorami podprogramów realizujących pewne popularne działania, które inni programiści mogą zestawiać (jak podstawowe cegiełki), by po uzupełnieniu swoimi podprogramami uzyskać gotowy produkt.

1.2.2. Programowanie strukturalne

Kolejnym paradygmatem, który można przedstawić jako dalsze udoskonalenie paradygmatu proceduralnego (a więc także i imperatywnego) jest *paradygmat strukturalny*. Programowanie strukturalne korzysta z bardzo ważnego wyniku informatyki teoretycznej [40], mówiącego o możliwości zapisania każdego algorytmu za pomocą elementarnych obliczeń połączonych ze sobą następującymi sposobami (strukturami):

— *sekwencja*, czyli kolejne wykonywanie czynności[7];
— *selekcja*, czyli wybór następnej drogi działania na podstawie jakiegoś warunku uzależnionego od stanu maszyny[8];
— *pętla „dopóki"*, czyli ściśle określony fragment algorytmu powtarzany przy ściśle określonych warunkach[9];
— *podprogram* pozwalający wydzielony podalgorytm zapisać, nazwać i wywoływać wielokrotnie[10] — z zastrzeżeniem, że podprogramy mają **dokładnie jeden *punkt wejścia*** oraz **dokładnie jeden *punkt wyjścia***.
— *rekurencja*, czyli definiowanie podprogramu za pomocą tego samego podprogramu.

Powyższa lista nie zawiera żadnych instrukcji skoku, co jest ogromną zaletą. Instrukcje skoku, bowiem — w szczególności niesławna instrukcja skoku bezwarunkowego `goto` — uważane są za szkodliwe [11] i w rzeczy

[7] W wielu językach operatorem sekwencyjnego złożenia czynności jest średnik. Ale w niektórych językach takie złożenie jest oznaczane znakiem nowej linii lub po prostu przez napisanie kolejnych instrukcji jedna po drugiej.

[8] Podstawowa forma selekcja w językach programowania zwykle jest zapisywana przez `if...else...`

[9] Pętla „dopóki" zwykle zapisywana jest w językach programowania przez `while...` W praktyce programistycznej spotyka się też inne pętle (czyli *iteracje*), ale wszystkie dadzą się zapisać jako specjalne przypadki pętli „dopóki".

[10] Dlatego właśnie paradygmat strukturalny rozpatrujemy jako podparadygmat programowania proceduralnego.

samej przyczyniają się do spadku czytelności kodu, a co za tym idzie, do mniejszej efektywności tworzenia i pielęgnowania (poprawiania, ulepszania, modyfikowania) oprogramowania. Co więcej, ograniczenie się do wypunktowanych powyższych pięciu konstrukcji pozwala na stosowanie *logiki Hoare'a* [14, 20, 21, 68] do — względnie prostego[11] — dowodzenia poprawności algorytmów strukturalnych, czyli ścisłego wykazywania, że robią dokładnie to, co miały robić.

Wiele ze współczesnych języków programowania — praktycznie wszystkie obecnie używane języki niedeklaratywne — pozwalają na użycie tego paradygmatu. Większość z nich jednak rozszerza ów paradygmat pozwalając na stosowanie różnego rodzaju *„skoków strukturalnych"*[12], na przykład:

— w wielu językach instrukcja `return` pozwala zakończyć wykonywanie podprogramu w różnych miejscach, a więc utworzyć wiele punktów wyjścia z jednego podprogramu;
— w C i językach pochodzących od C instrukcje `break` oraz `continue` pozwalają odpowiednio na wyskoczenie z pętli oraz przeskoczenie części jej bieżącego obrotu;
— mechanizmy przechwytywania wyjątków całkowicie zaburzają kolejność wykonywania instrukcji wynikającą ze struktur, w jakie te instrukcje zostały opakowane — oczywiście tylko w sytuacjach wyjątkowych (jak sama nazwa wskazuje);
— niektóre systemy operacyjne dostarczają funkcji systemowych (na przykład `longjump` w Linuksie; także obsługa sygnałów, przypominająca nieco obsługę wyjątków), których wywołanie może spowodować przekazanie sterowania praktycznie dowolnemu miejscu w programie — a więc realnie skok w dowolne miejsce programu;
— w końcu wiele języków dostarcza wprost instrukcję `goto`, choć czasem są nakładane pewne ograniczenia w jej stosowaniu.

Elementem programowania strukturalnego jest także pewna ważna technika (metodologia) programowania, mianowicie *top-down* [68]. Jest ona niejako odwróceniem techniki bottom-up (jak sama nazwa wskazuje). Polega ona mianowicie na dzieleniu całego zadania programistycznego na mniejsze podzadania zgodnie z przewidywaną strukturą na najwyższym poziomie, wypełnianiu tej struktury rozkazami elementarnymi tam, gdzie to możliwe, a następnie (tam gdzie nie można było jeszcze wstawić rozkazów elementarnych) zastosowaniu rekurencyjnie takiego dzielenia dalej, w głąb, do coraz to drobniejszych zadań.

[11] Prostego w porównaniu z próbą dowodzenia poprawności imperatywnego algorytmu zbudowanego niestrukturalnie, ze skokami różnego rodzaju. Dowodzenie poprawności takiego rodzaju algorytmów jest praktycznie niemożliwe.

[12] Właściwie to wyrażenie jest oksymoronem, ale słyszy się je gdzieniegdzie, więc i my je zastosujemy.

1.2.3. Programowanie obiektowe

Najbardziej rozpowszechnionym w dzisiejszych czasach paradygmatem programowania jest *paradygmat obiektowy*. Jest to właściwie paradygmat strukturalny rozszerzony jedynie o pojęcia *klas* i *obiektów*. Obiekty są tutaj zamkniętymi kontenerami zawierającymi dane (co przypomina rekordy czy też struktury znane z takich języków jak Pascal czy C), ale oprócz danych także podprogramy na tych danych działające, zwane *metodami*.

Formalnie rozszerzenie to jest niewielkie, ale pociąga za sobą daleko idące konsekwencje — na tyle poważne, że wielu teoretyków i praktyków programowania wręcz oddziela programowanie obiektowe od imperatywnego. My tak tego przedstawiać nie będziemy, bowiem program w języku obiektowym jest nadal sekwencją rozkazów. Jednakże te rozkazy — w języku czysto obiektowym — nie są wydawane maszynie jako całości, lecz **są wydawane poszczególnym obiektom** być może także przez inne obiekty[13].

Takie wydzielenie danych wraz z możliwymi do wykonania na nich czynnościami (w odróżnieniu od procedur, które są opakowaniem samych czynności, oraz w odróżnieniu od wspomnianych wyżej rekordów, które są opakowaniem samych danych) pozwala na wyróżnienie pewnych cech programowania obiektowego, których próżno szukać w innych paradygmatach imperatywnych:

— *hermetyzacja*, inaczej *enkapsulacja*, polegająca na tym, że tylko pewne dane i metody obiektu (stanowiące jego *interfejs*) są widoczne „na zewnątrz", dla innych obiektów; natomiast jego *implementacja* jest ukryta przed — umyślnym bądź przypadkowym — „uszkodzeniem" czy też złym wykorzystaniem;

— *dziedziczenie* pozwalające tworzyć obiekty bardziej skomplikowane na bazie prostszych; co więcej, dziedziczenie klas przekłada się na zawieranie się jednej w drugiej, a to oznacza, że obiekty mogą należeć jednocześnie do wielu klas, co ma istotne znaczenie dla polimorfizmu (poniżej);

— *abstrakcja danych* wynikająca bezpośrednio z hermetyzacji i dziedziczenia — można w prosty sposób definiować ogólne obiekty (czy też klasy), które są jedynie wzorcami pewnych bardziej skomplikowanych, doprecyzowanych obiektów;

— w końcu *polimorfizm dynamiczny* (inaczej *polimorfizm obiektowy*), który dzięki dziedziczeniu pozwala obiektom automatycznie dobierać odpowiednie metody do swojego aktualnego typu.

Dzięki tym wszystkim cechom programowanie obiektowe zyskało wielu zwolenników, powołujących się na podobieństwo modelu obiektowego do świata rzeczywistego [66] oraz do sposobu ludzkiego myślenia o otaczającej

[13] Obiekty mogą więc się w pewnym sensie porozumiewać.

rzeczywistości[14] i w dużej mierze dzięki językowi C++ zyskało ogromną popularność. Znalazło się jednak także wielu krytyków programowania obiektowego [4, 16, 31, 32]. Do tego wszystkiego wrócimy w Podrozdziale 4.2.

1.2.4. Programowanie funkcyjne

Funkcyjny paradygmat programowania jest podparadygmatem programowania deklaratywnego. W programowaniu funkcyjnym (także: *funkcjonalnym*) — tak jak w ogóle w programowaniu deklaratywnym — nie podajemy maszynie czynności do wykonania[15], ale opisujemy pożądany wynik: tutaj przy pomocy funkcji. W związku z tym, zadaniem interpretera lub kompilatora języka funkcyjnego jest wyłącznie obliczenie wartości funkcji głównej, a więc pewnego wyrażenia. Ewentualne wejście/wyjście programu jest jedynie *efektem ubocznym* wykonywania obliczeń.

W programowaniu czysto funkcyjnym mamy do czynienia ze ścisłą separacją funkcji czystych (*referencyjnie przezroczystych*), które zawsze przyjmują tę samą wartość dla tych samych argumentów (a więc nie zależą w żaden sposób od stanu maszyny, czy też jej „zewnętrza", to jest urządzeń wejścia/wyjścia, użytkownika, pamięci zewnętrznej...), od *akcji*, które mogą powodować i wykorzystywać efekty uboczne, gdy są wykonywane, ale nie są funkcjami w rozumieniu programowania funkcyjnego. Funkcja jest tutaj więc rozumiana całkowicie matematycznie — jako przyporządkowanie pewnych wartości pewnym argumentom.

W związku z pojęciem programu jako złożenia pewnych funkcji, w programowaniu funkcyjnym nie występują zmienne znane z programowania imperatywnego (bo są one abstrakcją stanu maszyny, do którego funkcja dostępu nie ma) ani tradycyjne pętle (potrzebujące do kontroli swojego działania dostępu do stanu maszyny), zamiast których używa się rekurencji[16].

Z drugiej strony, języki funkcyjne traktują funkcje jako *wartości pierwszego rzędu*, to jest mogą być one takimi samymi wartościami argumentów i wyników innych funkcji jak dane „prostsze" — liczby, napisy, listy...

Podstawy programowania funkcyjnego — a w szczególności *czysto funkcyjnego* — są także ściśle matematyczne: opiera się ono zwykle teoretycznie na *przeźroczystości referencyjnej* oraz *λ-rachunku* [7], który pozwala na wprowadzenie najściślejszej możliwej teoretycznie *kontroli typów* wraz z automatycznym o nich wnioskowaniem.

Powyższe cechy przyczyniają się do większej kontroli poprawności algorytmu (i pozwalają na jej łatwiejsze dowodzenie w razie potrzeby). Dzięki

[14] Jest w tym chyba jednak pewna przesada...

[15] Choć czasem może to wyglądać jak ciąg rozkazów...

[16] Która w językach funkcyjnych w postaci *rekurencji ogonowej* jest właściwie tak samo wydajna jak zwykła pętla.

ścisłej kontroli typów i separacji funkcji od akcji wzrasta także czytelność kodu i w sposób naturalny wsparta jest jego modularyzacja.

Ponadto przezroczystość referencyjna pozwala stosować *leniwe wartościowanie* wyniku funkcji, a więc obliczanie tylko tych fragmentów wyniku, które faktycznie są potrzebne innej funkcji/akcji. Ta leniwość natomiast pozwala na budowanie i wykorzystywanie struktur nieskończonych.

W końcu, przezroczystość referencyjna (w tym niezależność od efektów ubocznych) wraz z leniwym wartościowaniem umożliwia obliczanie składowych funkcji niezależnie od siebie, co w naturalny sposób pozwala na automatyczne zrównoleglanie kodu[17] i przetwarzanie potokowe.

W szczegółach programowaniem funkcyjnym zajmiemy się w Rozdziale 5.

1.2.5. Programowanie logiczne

W niniejszej pracy jeszcze jeden z paradygmatów zostanie szerzej odmówiony — *programowanie logiczne*, zwane też *programowaniem w logice* (Rozdział 6).

W programowaniu logicznym — analogicznie do programowania funkcyjnego (bo oba należą do paradygmatu deklaratywnego) — nie opisujemy wprost drogi do rozwiązania, lecz przedstawiamy maszynie zbiór pewnych zależności (przesłanek) oraz cel do sprawdzenia w formie pytania. W toku swojego działania maszyna ma za zadanie spróbować udowodnić podany cel na podstawie danych przesłanek. Wszystkie obliczenia czy też działania maszyny pojawiają się jako efekty uboczne owego dowodzenia.

Wydaję się, że jest to jeden z najwyższych poziomów abstrakcji w programowaniu: programista tutaj nie posługuje się w opisie problemu właściwie żadnymi czynnościami (a przecież i w programowaniu funkcyjnym można definicje funkcji utożsamić z opisem pewnych czynności), lecz określa jedynie zależności zachodzące pomiędzy elementami programu. Sam komputer ma za zadanie wyciągnąć odpowiednie wnioski z owych przesłanek.

1.2.6. Paradygmaty a języki

W Tabeli 1.1 prezentujemy kilka przykładów dość popularnych języków programowania wraz z przykładami głównych paradygmatów w nich reprezentowanych.

[17] A w dzisiejszych czasach, gdy niemal każdy komputer jest maszyną równoległą, jest to niebagatelna zaleta.

Tabela 1.1. Języki programowania a główne paradygmaty

języki	**paradygmaty**
asemblery, „stary" BASIC, „stary" Fortran	imperatywny proceduralny
„stary" Pascal, C	imperatywny proceduralny strukturalny
C++, Object Pascal, Ada	imperatywny proceduralny strukturalny obiektowy
Smalltalk, C#, Java	obiektowy
Lisp, Scheme, Logo, ML, OCaml	proceduralny funkcyjny
Haskell	czysto funkcyjny
Planner, Prolog	logiczny
Python, Ruby	proceduralny strukturalny obiektowy funkcyjny
SQL	deklaratywny (ale ani ściśle funkcyjny, ani ściśle logiczny)

1.3. Inne paradygmaty

Warto na koniec tego rozdziału wymienić jeszcze kilka bardziej niszowych paradygmatów, o których jednak wypadałoby słyszeć.

Większość z poniższych to miniparadygmaty, w tym sensie, że raczej współistnieją z „obszerniejszymi" paradygmatami (jak wymienione poprzednio) niż same stanowią rdzeń jakiegokolwiek języka programowania.

Programowanie modularne jest pośrednie między programowaniem obiektowym a proceduralnym. W tym paradygmacie główną jednostką planowania programu i jego tworzenia jest moduł (pakiet) zawarty zwykle w osobnym pliku i w wielu aspektach traktowany jako obiekt. Przykłady języków: Ada, Haskell, Python.

Programowanie aspektowe jest blisko związane z paradygmatem modularnym, bowiem jego celem jest ścisły podział problemu na jak najbardziej niezależne logicznie części i ograniczenie ich liczby styków oraz

ścisłe kontrolowanie każdego z nich [23, 28, 49]. Przykłady języków: AspectJ.

Programowanie komponentowe to kolejny paradygmat związany z modularyzacją programów, a jednocześnie z programowaniem obiektowym. Tutaj komponenty to jak najbardziej samodzielne obiekty wyposażone w ściśle wyspecyfikowany interfejs, wykonujące pewne określone usługi [18]. Zwykle paradygmat ten związany jest ściśle z programowaniem zdarzeniowym. Przykłady języków: Eiffel, Oberon.

Programowanie agentowe można uznać za nieco bardziej abstrakcyjną formę programowania obiektowego. Tutaj jednostką podstawową jest oczywiście agent [22], czyli wyspecjalizowany i odporny na błędy i niepowodzenia, a jednocześnie samodzielny obiekt, który w pewnym środowisku (często rozległym lub heterogenicznym, jak sieć komputerowa) może pracować sam, a w potrzebie komunikować się z innymi agentami. Działający w sieci agenci często dublują swoje czynności, po to, by zapewnić maksymalną odporność na błędy i utratę wyników. Nie bez znaczenia jest też ewentualna możliwość samoreplikacji agentów — w odpowiednim środowisku i warunkach[18]. Przykłady języków: JADE (framework Javy).

Programowanie zdarzeniowe (inaczej: *sterowane zdarzeniami*) to takie, gdy program składa się z wielu niezależnych podprogramów, których kolejność wykonania nie jest określona z góry przez program główny, lecz które są uruchamiane w reakcji na zaistnienie pewnych zdarzeń. Oprócz wspomnianych związków tego paradygmatu z komponentowym, obiektowym czy agentowym widać go w systemach operacyjnych, które działają praktycznie w oparciu o ten paradygmat. W reszcie, obsługa wyjątków w różnych językach ma charakter programowania zdarzeniowego.

Programowanie kontraktowe jest ściśle związane z paradygmatem obiektowym (ale mogłoby mieć również swoje miejsce jako rozszerzenie programowania strukturalnego) i polega na takim pisaniu kodu, by mógł być on automatycznie sprawdzony (pod względem zgodności ze specyfikacją) i ewentualnie przetestowany [36]. Przykłady języków: Eiffel, interfejsy w Javie.

Programowanie generyczne (inaczej: *uogólnione*, *rodzajowe*) umożliwia tworzenie jednostek (klas, obiektów, funkcji, typów) parametrycznych, lub inaczej mówiąc polimorficznych, uogólnionych, które stają się pełnoprawnymi jednostkami w chwili ich dookreślenia, co może zostać odłożone do momentu skorzystania z ich definicji w gotowym programie. Przykłady języków: Ada, C++, Haskell.

Programowanie refleksyjne pozwala na pisanie programów samomody-

[18] Pewnym przykładem niechlubnego zastosowania tego paradygmatu jest oczywiście plaga różnego rodzaju wirusów komputerowych...

fikujących się. Oznacza to, że program sam może „oglądać" własny kod, ale co ważniejsze, może też go modyfikować. Przykładem dość popularnego języka posiadającego mechanizm refleksji jest Python (i dlatego wrócimy też do tego mechanizmu w Podrozdziale 7.5.7). Inne przykłady takich języków: Lisp, Scheme.

Programowanie sterowane przepływem danych polega na konstruowaniu programów nie w oparciu o ustaloną kolejność czynności, lecz o dostępność danych i wykonywanie na nich czynności w czasie, gdy dane staną się dostępne. Przykłady tego paradygmatu to praca arkusza kalkulacyjnego (który przelicza dane, gdy tylko się zmienią) oraz przetwarzanie potokowe dobrze znane z Uniksowych systemów operacyjnych. Przykłady języków: Linda.

Programowanie współbieżne, równoległe, rozproszone to trzy ściśle ze sobą związane (choć nietożsame) paradygmaty, bliskie także programowaniu sterowanemu przepływem danych. Problemy tego paradygmatu związane są z podziałem czasu jednostki wykonującej rozkazy (lub jednostek) pomiędzy procesy, synchronizacją procesów, synchronizacją ich dostępów do pamięci wspólnej, przesyłaniem komunikatów pomiędzy procesami. Poświęcimy tym zagadnieniom cały Rozdział 8.

1.4. Pytania i zadania

1. Do czego służy logika Hoare'a?
2. Jakie dwie główne gałęzie paradygmatów wyróżniamy? Co je różni?
3. Wymień główne paradygmaty wraz z językami je reprezentującymi.
4. Czym charakteryzuje się architektura von Neumanna?
5. Skąd pochodzi nazwa języka programowania 'Ada'?
6. Jakie dwie główne techniki programowania związane są z paradygmatem proceduralnym i strukturalnym?
7. Jakie są własności stosu? Do czego służy stos maszynowy?
8. Dlaczego instrukcja `goto` jest uważana za szkodliwą?
9. Jakie instrukcje — nie licząc `goto` — w nowoczesnych językach imperatywnych zaburzają strukturalność?
10. Jakie cechy zdecydowały o popularności paradygmatu obiektowego?
11. Jakie struktury (w programowaniu strukturalnym) wystarczą do napisania każdego programu?
12. Na czym polega przezroczystość referencyjna? Co ona umożliwia?
13. Co jest podstawą programowania logicznego?
14. Do jakiego paradygmatu zaliczysz język SQL? Dlaczego?
15. Dlaczego w asemblerze czy też w kodzie maszynowym nie ma ograniczenia liczby punktów wejścia ani liczby punktów wyjścia podprogramu?

Rozdział 2

Składnia i semantyka języków programowania

2.1. Składnia a semantyka

Opis języków — formalnych (takimi są języki programowania) [2, 54, 63], ale także nieformalnych (jak języki naturalne) — składa się zawsze z dwóch części:
— *składni* (inaczej: gramatyki, syntaksy), czyli opisu poprawnej budowy wypowiedzeń w tym języku;
— *semantyki*, czyli opisu znaczenia poszczególnych wypowiedzeń języka.
Zarówno składnia jak i semantyka najdawniejszych języków programowania były opisywane nieformalnie (lub nawet wcale nie były opisywane — za cały opis wystarczał działający kompilator lub interpreter języka). Sposób ten — wraz ze wzrostem skomplikowania coraz to nowszych języków — szybko stał się niewystarczający, bowiem na opisie niesformalizowanym ciąży zawsze niejednoznaczność, na którą w programowaniu nie można sobie pozwolić.

2.2. Składnia

Składnia dzisiejszych języków programowania opisywana jest całkowicie formalnie [17, 53]. Odbywa się to zwykle w trzech etapach:
— określenie *alfabetu*, czyli zbioru znaków, którymi można posługiwać się w danym języku[1];
— określenie zbioru poprawnych leksemów i ich podział na podzbiory — za pomocą *gramatyk regularnych*;
— określenie reguł budowania poprawnych składowych języka (definicji, deklaracji, wyrażeń, instrukcji, w końcu całych programów) z poszczególnych leksemów — za pomocą *gramatyk bezkontekstowych*.
Właściwie jest jeszcze czwarty etap — opisywanie zależności kontekstowych — leżący na pograniczu składni i semantyki, bo nie wszystko da się uchwycić w poprzednich trzech etapach. Jednakże, rozważać będziemy go tu osobno.

2.2.1. Leksemy i wyrażenia regularne

Leksem (także: *token*) to najmniejsza część języka programowania, która logicznie nie może być podzielona na mniejsze kawałki. Leksemami są na przykład identyfikatory (`X`, `a1`, `cout`, `printf`), słowa kluczowe (`if`, `while`), literały różnego rodzaju (`42`, `0.31415e1`, `0xf00`, `"jakiś napis"`), operatory (`<<`, `+`).

Najpowszechniej używanym (bo i najwygodniejszym) sposobem określenia zbioru poprawnych leksemów i ich podziału na poszczególne podzbiory

[1] W przypadku języków programowania często ów alfabet pokrywa się z zestawem znaków określonym przez wybrane (przez projektantów języka) kodowanie.

(a więc czy dany leksem jest identyfikatorem, literałem całkowitym, słowem kluczowym itd.) są gramatyki regularne lub równoważne im *wyrażenia regularne*. Zdefiniujemy tutaj podstawową formę wyrażeń regularnych.

Definicja 2.1. Zbiór wyrażeń regularnych nad alfabetem Σ oznaczać będziemy przez $\mathbf{regexp}(\Sigma)$. Każdy z poniższych napisów jest elementem zbioru $\mathbf{regexp}(\Sigma)$, czyli wyrażeniem regularnym nad alfabetem Σ:

$$\varepsilon \tag{2.1}$$

$$a \tag{2.2}$$

$$\zeta^* \tag{2.3}$$

$$\zeta\eta \tag{2.4}$$

$$\zeta|\eta \tag{2.5}$$

$$(\zeta) \tag{2.6}$$

(dla każdego $a \in \Sigma$, $\zeta \in \mathbf{regexp}(\Sigma)$, $\eta \in \mathbf{regexp}(\Sigma)$), gdzie ε oznacza napis pusty.

Każde z wyrażeń regularnych generuje pewien zbiór napisów[2] nad alfabetem Σ — zbiór generowany przez wyrażenie ζ oznaczać będziemy przez $\mathcal{L}_\zeta$. Jakie zbiory generują poszczególne wyrażenia regularne? Żeby dobrze określić te zbiory, potrzebujemy pewnych dodatkowych oznaczeń i definicji.

Definicja 2.2. Dla danych dowolnych zbiorów A oraz B złożonych z napisów określamy

$$AB = \{\zeta\eta : \zeta \in A \wedge \eta \in B\} \tag{2.7}$$

(słownie: zbiór AB składa się ze wszystkich możliwych konkatenacji parami napisów ze zbiorów A oraz B w tej właśnie kolejności);

$$A^0 = \{\varepsilon\} \quad \text{(zbiór złożony z napisu pustego)}, \tag{2.8}$$

$$A^n = A^{n-1}A \quad \text{dla } n \in \mathbb{N} \setminus \{0\} \tag{2.9}$$

(słownie: zbiór A^n składa się ze wszystkich możliwych konkatenacji dowolnych n napisów ze zbioru A);

$$A^* = \bigcup_{i=0}^{\infty} A^i \tag{2.10}$$

(słownie: zbiór A^* składa się ze wszystkich możliwych skończonych konkatenacji dowolnej liczby [także zera] napisów ze zbioru A).

[2] Zbiór napisów to inaczej właśnie *język* — tutaj język generowany przez dane wyrażenie regularne.

Przykład 2.3. Rozważmy zbiory $A = \{\texttt{aa}\}$ oraz $B = \{\texttt{b}, \texttt{c}\}$. Wtedy:

$$AB = \{\texttt{aab}, \texttt{aac}\}, \tag{2.11}$$
$$BA = \{\texttt{baa}, \texttt{caa}\}, \tag{2.12}$$
$$A^3 = \{\texttt{aaaaaa}\}, \tag{2.13}$$
$$B^2 = \{\texttt{bb}, \texttt{bc}, \texttt{cb}, \texttt{cc}\}, \tag{2.14}$$
$$A^* = \{\varepsilon, \texttt{aa}, \texttt{aaaa}, \texttt{aaaaaa}, \ldots\} \tag{2.15}$$
(wszystkie napisy złożone z parzystej liczby znaków a).

Teraz możemy zdefiniować, jakie zbiory generują poszczególne wyrażenia regularne.

Definicja 2.4.

$$\mathcal{L}_\varepsilon = \{\varepsilon\} \quad \text{(zbiór złożony z napisu pustego)}, \tag{2.16}$$
$$\mathcal{L}_a = \{a\} \quad \text{(zbiór złożony z napisu jednoznakowego } a\text{)}, \tag{2.17}$$
$$\mathcal{L}_{\zeta^*} = (\mathcal{L}_\zeta)^*, \tag{2.18}$$
$$\mathcal{L}_{\zeta\eta} = \mathcal{L}_\zeta \mathcal{L}_\eta, \tag{2.19}$$
$$\mathcal{L}_{\zeta|\eta} = \mathcal{L}_\zeta \cup \mathcal{L}_\eta, \tag{2.20}$$
$$\mathcal{L}_{(\zeta)} = \mathcal{L}_\zeta \quad \text{(nawiasy służą tylko do grupowania)}. \tag{2.21}$$

(dla każdego $a \in \Sigma$, $\zeta \in \mathbf{regexp}(\Sigma)$, $\eta \in \mathbf{regexp}(\Sigma)$).

Ponieważ wyrażenia regularne opisują (jak widać powyżej) pewne działania na zbiorach, przyjmuje się priorytety operacji (dla $\zeta, \eta, \vartheta \in \mathbf{regexp}(\Sigma)$):

$$\zeta\eta|\vartheta = (\zeta\eta)|\vartheta, \tag{2.22}$$
$$\zeta|\eta^* = \zeta|(\eta^*), \tag{2.23}$$
$$\zeta\eta^* = \zeta(\eta^*). \tag{2.24}$$

Innymi słowy: najsilniejsze jest (2.3)/(2.18), potem (2.4)/(2.19), na końcu (2.5)/(2.20).

Przykład 2.5. Rozważmy alfabet $\Sigma = \{\texttt{a}, \texttt{b}\}$. Wtedy:

$$\mathcal{L}_{\texttt{a}} = \{\texttt{a}\}, \tag{2.25}$$
$$\mathcal{L}_{\texttt{ab}^*} = \{\texttt{a}, \texttt{ab}, \texttt{abb}, \texttt{abbb}, \ldots\}, \tag{2.26}$$
$$\mathcal{L}_{\texttt{a}|\texttt{b}} = \Sigma, \tag{2.27}$$
$$\mathcal{L}_{\texttt{a}|\texttt{b}^*} = \{\varepsilon, \texttt{a}, \texttt{b}, \texttt{bb}, \texttt{bbb}, \ldots\}, \tag{2.28}$$
$$\mathcal{L}_{\texttt{a}^*|\texttt{b}^*} = \{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \ldots, \texttt{b}, \texttt{bb}, \texttt{bbb}, \ldots\}, \tag{2.29}$$
$$\mathcal{L}_{(\texttt{a}|\texttt{b})^*} = \Sigma^*, \tag{2.30}$$

$$\begin{aligned}\mathcal{L}_{\texttt{a*b*}} &= \{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \ldots, \\ &\quad \texttt{b}, \texttt{ab}, \texttt{aab}, \texttt{aaab}, \ldots, \\ &\quad \texttt{bb}, \texttt{abb}, \texttt{aabb}, \texttt{aaabb}, \ldots, \\ &\quad \texttt{bbb}, \texttt{abbb}, \texttt{aabbb}, \texttt{aaabbb}, \ldots, \\ &\quad \ldots\}.\end{aligned} \tag{2.31}$$

W praktyce programistycznej wyrażenia regularne wzbogacane są o pewne skróty, które nie uogólniają poprzednich definicji, ale przyczyniają się do zwartości zapisu. Wybrane definiujemy poniżej.

Definicja 2.6.

$$\zeta^+ = \zeta\zeta^*, \tag{2.32}$$

$$[a_1 a_2 \ldots a_n] = (a_1|a_2|\ldots|a_n), \tag{2.33}$$

$$\begin{aligned}[a_1 - a_n] &= (a_1|a_2|\ldots|a_n) \\ &\quad \text{(o ile w } \Sigma \text{ określono porządek} \\ &\quad \text{i } a_1, a_2, \ldots, a_n \text{ są kolejno, bez przerw).}\end{aligned} \tag{2.34}$$

Skróty (2.33) oraz (2.34) można łączyć, na przykład:

$$[a_1 - a_n b c_1 - c_m] = (a_1|a_2|\ldots|a_n|b|c_1|c_2|\ldots|c_m) \tag{2.35}$$

(wszystko to dla każdego $a_i, b, c_j \in \Sigma$, $\zeta \in \mathbf{regexp}(\Sigma)$).

Może na koniec jeszcze drobna uwaga: w praktyce informatycznej/programistycznej nie używa się zwykle różnych krojów czcionek, więc gdy zachodzi potrzeba użycia znaku mającego znaczenie specjalne w wyrażeniach regularnych (jak $+$, $*$, []) w swoim zwykłym znaczeniu (jako elementu alfabetu), wtedy poprzedzamy go (zgodnie z powszechną tradycją) odwrotnym ukośnikiem \.

Mając wyrażenia regularne, projektanci języka opisują nimi i klasyfikują leksemy występujące w opisywanym języku. Można na przykład teraz zdefiniować kilka zbiorów leksemów ze znanych nam języków.

Przykład 2.7. Rozważmy alfabet Σ zawierający znaki odpowiadające kodowaniu ASCII wraz z odpowiednim porządkiem. Możemy wtedy zdefiniować różne zbiory leksemów z języka C:

— liczby całkowite dziesiętne bez znaku: $\mathcal{L}_{\texttt{[1-9][0-9]*}}$;
— liczby całkowite ósemkowe bez znaku: $\mathcal{L}_{\texttt{0[0-7]*}}$;
— liczby całkowite szesnastkowe bez znaku: $\mathcal{L}_{\texttt{0[xX][0-9a-fA-F]+}}$;
— identyfikatory: $\mathcal{L}_{\texttt{[a-zA-Z_][a-zA-Z_0-9]*}}$.

2.2.2. Gramatyki bezkontekstowe

Kolejny poziom składni języka programowania stanowią jego struktury (wyrażenia, instrukcje itp.). Składają się one oczywiście z poprawnych leksemów, ale co więcej, ustawionych względem siebie w pewien określony sposób. Tę strukturę definiujemy zwykle za pomocą *gramatyk bezkontekstowych*.

Definicja 2.8. Gramatyką bezkontekstową (dokładniej: generatywną gramatyką bezkontekstową) nazywamy czwórkę $\mathcal{G} = (T, N, S, P)$, spełniającą warunki:

$$T \neq \emptyset, \tag{2.36}$$
$$S \in N, \tag{2.37}$$
$$T \cap N = \emptyset, \tag{2.38}$$
$$P \subset N \times (T \cup N)^*. \tag{2.39}$$

Słowami można wyrazić powyższe wzory następująco:
— (2.36): zbiór T (zwany zbiorem *symboli terminalnych*, które to symbole możemy utożsamić z leksemami/tokenami języka) nie jest zbiorem pustym;
— (2.37): zbiór N (zwany zbiorem *symboli nieterminalnych*, które to symbole możemy traktować jako pewne symbole pomocnicze) nie jest zbiorem pustym i zawiera co najmniej jeden symbol S (zwany *symbolem startowym*);
— (2.38): zbiory symboli terminalnych i nieterminalnych nie mają elementów wspólnych (inaczej: każdy używany symbol jest jednoznacznie określony — terminalny albo nieterminalny);
— (2.39): zbiór P to zbiór par uporządkowanych postaci (A, ζ), gdzie $A \in N$ (czyli A jest pojedynczym symbolem nieterminalnym) oraz $\zeta \in (T \cup N)^*$ (czyli ζ jest dowolnym, być może pustym, napisem złożonym z symboli terminalnych i/lub nieterminalnych).
Pary należące do zbioru P nazywamy *produkcjami* i zapisujemy zwykle: $A \to \zeta$ zamiast (A, ζ).

Jak taka gramatyka definiuje język? Otóż, żeby dobrze opisać język generowany przez daną gramatykę bezkontekstową $\mathcal{G}$ (a oznaczany $\mathcal{L}_\mathcal{G}$), potrzebujemy jeszcze kilku definicji.

Definicja 2.9. Rozważmy gramatykę $\mathcal{G} = (T, N, S, P)$ oraz dwa napisy (być może puste) złożone z symboli terminalnych i/lub nieterminalnych $\zeta, \eta \in (T \cup N)^*$. Mówimy, że napis η jest *bezpośrednio wyprowadzalny* z napisu ζ (i zapisujemy ten fakt $\zeta \underset{\mathcal{G}}{\Rightarrow} \eta$) wtedy i tylko wtedy, gdy istnieją napisy

$\alpha, \beta, \gamma \in (T \cup N)^*$, symbol nieterminalny $A \in N$ oraz produkcja $A \to \gamma \in P$ takie, że

$$\zeta = \alpha A \beta, \qquad \eta = \alpha \gamma \beta. \tag{2.40}$$

Definicja 2.10. Dla gramatyki $\mathcal{G} = (T, N, S, P)$ oraz dwóch napisów (być może pustych) złożonych z symboli terminalnych i/lub nieterminalnych $\zeta, \eta \in (T \cup N)^*$, mówimy, że napis η jest *(pośrednio) wyprowadzalny* z napisu ζ (i zapisujemy ten fakt $\zeta \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \eta$) wtedy i tylko wtedy, gdy spełniony jest jeden z następujących warunków:

$$\zeta \underset{\mathcal{G}}{\Rightarrow} \eta \tag{2.41}$$

lub

$$\bigvee_{\alpha \in (T \cup N)^*} \left(\zeta \underset{\mathcal{G}}{\Rightarrow} \alpha \wedge \alpha \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \eta \right). \tag{2.42}$$

Innymi słowy: napis η jest (pośrednio) wyprowadzalny z napisu ζ wtedy i tylko wtedy, gdy istnieje ciąg napisów $\alpha_1, \ldots, \alpha_n \in (T \cup N)^*$ taki, że $\zeta = \alpha_1$ oraz $\eta = \alpha_n$ i jednocześnie:

$$\bigwedge_{i=1,\ldots,n-1} \alpha_i \underset{\mathcal{G}}{\Rightarrow} \alpha_{i+1}. \tag{2.43}$$

Definicja 2.11. Językiem $\mathcal{L}_\mathcal{G}$ generowanym przez daną gramatykę bezkontekstową $\mathcal{G} = (T, N, S, P)$ nazywamy zbiór wszystkich napisów (nie wyłączając z góry napisu pustego) wyprowadzalnych (pośrednio) z symbolu startowego S, które składają się jedynie z symboli terminalnych. Inaczej:

$$\mathcal{L}_\mathcal{G} = \left\{ \zeta \in T^* : S \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \zeta \right\}. \tag{2.44}$$

Przykład 2.12. Zdefiniujmy za pomocą gramatyki bezkontekstowej $\mathcal{G} = (T, N, S, P)$ język $\mathcal{L}_\mathcal{G}$ poprawnych wyrażeń algebraicznych (czy też arytmetycznych) na trzech zmiennych o nazwach `x`, `y`, `z`, w których to wyrażeniach możemy używać dwuargumentowych znaków działań `+`, `-`, `*`, `/` oraz nawiasów okrągłych `(`, `)`. Zaczniemy od określenia zbioru symboli terminalnych:

$$T = \left\{ \texttt{x}, \texttt{y}, \texttt{z}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{(}, \texttt{)} \right\}. \tag{2.45}$$

Wyliczenie elementów zbioru N odłożymy, bo wyczytamy je po prostu z produkcji — symbole nieterminalne są tylko pomocniczymi symbolami występującymi właśnie w produkcjach. Jeden wyjątek: przyjmijmy, że S jest nieterminalnym symbolem startowym. A oto i produkcje:

$$S \;\to\; \mathit{Zmienna} \tag{2.46}$$

$$S \rightarrow S \quad Operator \quad S \tag{2.47}$$
$$S \rightarrow \texttt{(}S\texttt{)} \tag{2.48}$$
$$Zmienna \rightarrow \texttt{x} \tag{2.49}$$
$$Zmienna \rightarrow \texttt{y} \tag{2.50}$$
$$Zmienna \rightarrow \texttt{z} \tag{2.51}$$
$$Operator \rightarrow \texttt{+} \tag{2.52}$$
$$Operator \rightarrow \texttt{-} \tag{2.53}$$
$$Operator \rightarrow \texttt{*} \tag{2.54}$$
$$Operator \rightarrow \texttt{/} \tag{2.55}$$

Teraz widać, że zbiór symboli nieterminalnych to:

$$N = \Big\{S, Operator, Zmienna\Big\}. \tag{2.56}$$

Oczywiście, zarówno *Operator*, jak i *Zmienna* są tu pojedynczymi symbolami.

Z powyższych produkcji możemy też wyczytać pewną interpretację w języku potocznym. Mianowicie, nasze wyrażenie (S) może być
— pojedynczą zmienną (2.46),
— dwoma wyrażeniami z operatorem między nimi (2.47),
— wyrażeniem ujętym w nawiasy okrągłe (2.48).
Z kolei zmienną może być `x`, `y` lub `z` (2.49)–(2.51), operatorem zaś `+`, `-`, `*` lub `/` (2.52)–(2.55).

Jak teraz sprawdzić, że jakiś napis — powiedzmy: `x+y*(x)` — jest poprawnym napisem w naszym języku $\mathcal{L}_\mathcal{G}$? Wystarczy znaleźć dowolne wyprowadzenie pośrednie tego napisu z symbolu startowego S:

$$\begin{aligned}
S &\underset{\mathcal{G}}{\Rightarrow} S \quad Operator \quad S && \text{(z produkcji (2.47))} \\
&\underset{\mathcal{G}}{\Rightarrow} S \quad Operator \quad S \quad Operator \quad S && \text{(z (2.47))} \\
&\underset{\mathcal{G}}{\Rightarrow} S \quad \texttt{+} \quad S \quad Operator \quad S && \text{(z (2.52))} \\
&\underset{\mathcal{G}}{\Rightarrow} S \quad \texttt{+} \quad S \quad \texttt{*} \quad S && \text{(z (2.54))} \\
&\underset{\mathcal{G}}{\Rightarrow} S \quad \texttt{+} \quad S \quad \texttt{*(} \quad S \quad \texttt{)} && \text{(z (2.48))} \\
&\underset{\mathcal{G}}{\Rightarrow} Zmienna \quad \texttt{+} \quad S \quad \texttt{*(} \quad S \quad \texttt{)} && \text{(z (2.46))} \\
&\underset{\mathcal{G}}{\Rightarrow} Zmienna \quad \texttt{+} \quad Zmienna \quad \texttt{*(} \quad S \quad \texttt{)} && \text{(z (2.46))} \\
&\underset{\mathcal{G}}{\Rightarrow} Zmienna \quad \texttt{+} \quad Zmienna \quad \texttt{*(} \quad Zmienna \quad \texttt{)} && \text{(z (2.46))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+} \quad Zmienna \quad \texttt{*(} \quad Zmienna \quad \texttt{)} && \text{(z (2.49))}
\end{aligned} \tag{2.57}$$

$\underset{\mathcal{G}}{\Rightarrow}$ `x+y*(` *Zmienna* `)` (z (2.50))

$\underset{\mathcal{G}}{\Rightarrow}$ `x+y*(x)` (z (2.49))

Albo inne (może być wiele różnych wyprowadzeń):

$$
\begin{aligned}
S &\underset{\mathcal{G}}{\Rightarrow} S\ \mathit{Operator}\ S && \text{(z produkcji (2.47))} \\
&\underset{\mathcal{G}}{\Rightarrow} \mathit{Zmienna}\ \mathit{Operator}\ S && \text{(z (2.46))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x}\ \mathit{Operator}\ S && \text{(z (2.49))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+}\ S && \text{(z (2.52))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+}\ S\ \mathit{Operator}\ S && \text{(z (2.47))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+}\ \mathit{Zmienna}\ \mathit{Operator}\ S && \text{(z (2.46))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+y}\ \mathit{Operator}\ S && \text{(z (2.50))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+y*}\ S && \text{(z (2.54))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+y*(}\ S\ \texttt{)} && \text{(z (2.48))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+y*(}\ \mathit{Zmienna}\ \texttt{)} && \text{(z (2.46))} \\
&\underset{\mathcal{G}}{\Rightarrow} \texttt{x+y*(x)} && \text{(z (2.49))}
\end{aligned}
\tag{2.58}
$$

Przykład 2.13. Rozważmy jeszcze fragment gramatyki bezkontekstowej $\mathcal{G}$, poświęcony definicji pewnych instrukcji w pewnym języku programowania. Oto wybrane produkcje:

$$
\begin{aligned}
\mathit{Ins} &\to \mathit{InsWar} && (2.59)\\
\mathit{Ins} &\to \mathit{InsSekw} && (2.60)\\
\mathit{Ins} &\to \mathit{InsPodst} && (2.61)\\
\mathit{InsWar} &\to \texttt{if}\ \texttt{(}\ \mathit{Wyr}\ \texttt{)}\ \mathit{Ins} && (2.62)\\
\mathit{InsWar} &\to \texttt{if}\ \texttt{(}\ \mathit{Wyr}\ \texttt{)}\ \mathit{Ins}\ \texttt{else}\ \mathit{Ins} && (2.63)\\
\mathit{InsSekw} &\to \texttt{\{}\ \mathit{SekwIns}\ \texttt{\}} && (2.64)\\
\mathit{SekwIns} &\to && (2.65)\\
\mathit{SekwIns} &\to \mathit{Ins}\ \mathit{SekwIns} && (2.66)\\
\mathit{InsPodst} &\to \mathit{LWart}\ \texttt{=}\ \mathit{Wyr}\ \texttt{;} && (2.67)
\end{aligned}
$$

Oczywiście (podobnie jak w Przykładzie 2.12) symbolami nieterminalnymi są:
$Ins, InsWar, InsSek, InsPodst, Wyr, SekwIns, LWart,$

natomiast symbolami terminalnymi:
`if`, `(`, `)`, `{`, `}`, `=`, `;`, `else`.
Ponieważ mamy do czynienia z fragmentem definicji języka — nie muszą to być naturalnie wszystkie występujące w tej gramatyce symbole terminalne i nieterminalne.

Warto zwrócić uwagę na produkcje opisujące sekwencję instrukcji ($SekwIns$). Pierwsza z nich (2.65) ma pustą prawą stronę (można także oznaczyć ten fakt symbolem napisu pustego ε) i nie jest to pomyłką — po prostu sekwencja instrukcji może nie zawierać żadnej instrukcji. Druga z tych produkcji (2.66) to rekurencyjna definicja sekwencji. Z tych dwóch produkcji wynika, że sekwencja instrukcji może składać się z dowolnej skończonej liczby instrukcji (od zera wzwyż).

Jak nietrudno się przekonać, poprawną (z dokładnością do znaków białych[3]) instrukcją (Ins) jest:

```
{
  if (a > b) {
    max = a;
    min = b;
  }
  else {
    max = b;
    min = a;
  }
}
```

Pod warunkiem, wszakże, że można gdzieś z dalszej części definicji tegoż języka wyprowadzić:

$$Wyr \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \texttt{a > b} \tag{2.68}$$

$$Wyr \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \texttt{a} \tag{2.69}$$

$$Wyr \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \texttt{b} \tag{2.70}$$

$$LWar \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \texttt{max} \tag{2.71}$$

$$LWar \overset{*}{\underset{\mathcal{G}}{\Rightarrow}} \texttt{min} \tag{2.72}$$

[3] Zwykle języki programowania ignorują znaki białe (spacje, tabulatory, znaki nowego wiersza) traktując je co najwyżej jako separatory tokenów (nie zawsze konieczne). Istnieją jednak wyjątki (z języków głównego nurtu: Haskell, Rozdział 5 i Python, Rozdział 7), które w pewnych miejscach używają znaków białych (w postaci wcięć) do oznaczania bloków programu.

2.2.2.1. Ułatwienia notacyjne

Tradycyjna (matematyczna) notacja gramatyk bezkontekstowych (jakiej używaliśmy powyżej) ma pewne wady:
— jest rozwlekła, mało zwarta; dałoby się parę logicznych i naturalnych skrótów opracować...
— nie pozwala wprost zapisywać powtórzeń, lecz zmusza do użycia — nie zawsze wygodnej — rekurencji (jak w produkcjach (2.65)–(2.66));
— zwykle symbole terminalne od nieterminalnych odróżnia się krojem czcionki (na przykład kursywa dla nieterminalnych, czcionka maszynowa dla terminalnych) lub wielkością liter (małą literą symbole terminalne, wielką — nieterminalne); taka konwencja nie jest ani przyjęta ani praktyczna w świecie informatycznym, gdzie często (na przykład w językach programowania) używa się jednego kroju czcionki, a symbole terminalne (jako elementy opisywanego języka) mogą zawierać dowolne znaki...

Wprowadźmy więc kilka ulepszeń i sprawdźmy jak będą wyglądać produkcje z Przykładu 2.13 ze strony 21.

Zaczniemy od wyraźniejszego rozdzielenia symboli terminalnych od nieterminalnych przez zapisanie symboli terminalnych w cudzysłowach `"..."` (w razie potrzeby możemy dzięki temu w symbolach terminalnych używać w naturalny sposób spacji):

$$
\begin{aligned}
Ins &\to InsWar \\
Ins &\to InsSekw \\
Ins &\to InsPodst \\
InsWar &\to \texttt{"if"}\ \texttt{"("}\ Wyr\ \texttt{")"}\ Ins \\
InsWar &\to \texttt{"if"}\ \texttt{"("}\ Wyr\ \texttt{")"}\ Ins\ \texttt{"else"}\ Ins \\
InsSekw &\to \texttt{"\{"}\ SekwIns\ \texttt{"\}"} \\
SekwIns &\to \\
SekwIns &\to Ins\ SekwIns \\
InsPodst &\to LWart\ \texttt{"="}\ Wyr\ \texttt{";"}
\end{aligned}
$$

Można w tym momencie zrezygnować ze stosowania różnych krojów do symboli terminalnych i pozostałych elementów definicji, bo symbole terminalne wyróżnione są przez cudzysłowy. My jednak zachowamy to rozróżnienie, ale tylko i wyłącznie dla czytelności — znaczenia już w tej chwili ono nie ma.

Kolejnym dodatkiem w naszej notacji będzie zapisanie kilku produkcji o identycznej lewej stronie w formie jednej produkcji i „alternatywy” (symbol |) po prawej stronie produkcji:

$$
\begin{aligned}
Ins &\to InsWar \mid InsSekw \mid InsPodst \\
InsWar &\to \texttt{"if"}\ \texttt{"("}\ Wyr\ \texttt{")"}\ Ins \mid
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & \texttt{"if"}\ \ \texttt{"("}\ \ Wyr\ \ \texttt{")"}\ \ Ins\ \ \texttt{"else"}\ \ Ins \\
InsSekw & \to & \texttt{"\{"}\ \ SekwIns\ \ \texttt{"\}"} \\
SekwIns & \to & \quad \mid\ Ins\ \ SekwIns \\
InsPodst & \to & LWart\ \ \texttt{"="}\ \ Wyr\ \ \texttt{";"}
\end{array}
$$

Druga z powyższych produkcji mogłaby być zapisana w jednej linii, ale możemy produkcje dzielić na wiele wierszy, nie wpływa to na ich sens. Ponadto w czwartej z produkcji mamy pustą część alternatywy — bo sekwencja instrukcji może być pusta lub składać się z instrukcji, po której następuje dalsza sekwencja instrukcji.

Możemy też wprowadzić nawiasy grupujące ( ) tak, by symbolu | można było używać do części prawej strony produkcji, nie do całej:

$$
\begin{array}{rcl}
Ins & \to & InsWar \mid InsSekw \mid InsPodst \\
InsWar & \to & \texttt{"if"}\ \ \texttt{"("}\ \ Wyr\ \ \texttt{")"}\ \ Ins\ \ (\ \mid\ \texttt{"else"}\ \ Ins\ ) \\
InsSekw & \to & \texttt{"\{"}\ \ SekwIns\ \ \texttt{"\}"} \\
SekwIns & \to & \quad \mid\ Ins\ \ SekwIns \\
InsPodst & \to & LWart\ \ \texttt{"="}\ \ Wyr\ \ \texttt{";"}
\end{array}
$$

Wprowadzimy także nawiasy wystąpienia opcjonalnego [ ], które mówią, że ciąg symboli dany w nich może, ale nie musi wystąpić:

$$
\begin{array}{rcl}
Ins & \to & InsWar \mid InsSekw \mid InsPodst \\
InsWar & \to & \texttt{"if"}\ \ \texttt{"("}\ \ Wyr\ \ \texttt{")"}\ \ Ins\ \ [\ \texttt{"else"}\ \ Ins\ ] \\
InsSekw & \to & \texttt{"\{"}\ \ SekwIns\ \ \texttt{"\}"} \\
SekwIns & \to & [\ Ins\ \ SekwIns\ ] \\
InsPodst & \to & LWart\ \ \texttt{"="}\ \ Wyr\ \ \texttt{";"}
\end{array}
$$

Zauważmy, że zapis [ζ] jest równoważny zapisowi (| ζ).

Na koniec ostatni dodatek: nawiasy powtórzeń { } , których zawartość może w ogóle nie wystąpić w wygenerowanym napisie, ale może też wystąpić raz lub więcej razy.

$$
\begin{array}{rcl}
Ins & \to & InsWar \mid InsSekw \mid InsPodst \\
InsWar & \to & \texttt{"if"}\ \ \texttt{"("}\ \ Wyr\ \ \texttt{")"}\ \ Ins\ \ [\ \texttt{"else"}\ \ Ins\ ] \\
InsSekw & \to & \texttt{"\{"}\ \ SekwIns\ \ \texttt{"\}"} \\
SekwIns & \to & \{\ Ins\ \} \\
InsPodst & \to & LWart\ \ \texttt{"="}\ \ Wyr\ \ \texttt{";"}
\end{array}
$$

W ten sposób pozbywamy się z zapisu rekurencji. Ponadto, ponieważ symbol *SekwIns* był nam tutaj tylko i wyłącznie potrzebny do wprowadzenia

rekurencji, możemy pozbyć się go i czwartą produkcję z powyższej listy podstawić do trzeciej:

$$\begin{aligned} Ins &\rightarrow InsWar \mid InsSekw \mid InsPodst \\ InsWar &\rightarrow \texttt{"if"}\ \texttt{"("}\ Wyr\ \texttt{")"}\ Ins\ [\ \texttt{"else"}\ Ins\ ] \\ InsSekw &\rightarrow \texttt{"\{"}\ \{\ Ins\ \}\ \texttt{"\}"} \\ InsPodst &\rightarrow LWart\ \texttt{"="}\ Wyr\ \texttt{";"} \end{aligned}$$

Zapis składający się z dziewięciu wierszy (Przykład 2.13, strona 21) skróciliśmy do czterech wierszy (a więc o ponad połowę!) dzięki dodatkowym oznaczeniom i uczyniliśmy go bardziej czytelnym a także niewrażliwym na zastosowaną czcionkę. Z drugiej strony warto podkreślić, że nowa notacja nie jest w sensie matematycznym mocniejsza ani słabsza od tradycyjnej notacji gramatyk bezkontekstowych (czyli jest im dokładnie równoważna; nie można zdefiniować więcej ani mniej języków) i zawsze możemy do tradycyjnej notacji wrócić (zamiast nawiasów klamrowych wprowadzając rekurencję i ewentualnie pomocnicze symbole nieterminalne, natomiast pozostałe skróty oddając przez powtórzenie produkcji z różnymi wariantami prawej strony).

Notacja, do której doszliśmy pokrywa się właściwie z *notacją Niklausa Wirtha* [67]. Różnica jest jedynie taka, że zamiast strzałki $\rightarrow$ Wirth stosował znak równości $=$, a każdą produkcję kończył kropką (dla wyraźnego oddzielenia od kolejnej, szczególnie jeśli produkcje rozciągały się na wiele wierszy). Nasz przykład w notacji Wirtha wyglądałby następująco (tu już stosujemy jedną czcionkę, bo i tak zapisywał to Wirth):

```
Ins      =  InsWar  |  InsSekw  |  InsPodst  .
InsWar   =  "if"  "("  Wyr  ")"  Ins  [  "else"  Ins  ]  .
InsSekw  =  "{"  {  Ins  }  "}"  .
InsPodst =  LWart  "="  Wyr  ";"  .
```

Notacja Wirtha bardzo podobna jest różnym wariantom BNF — *notacji Backusa-Naura* [5]. Nie będziemy się nią zajmować (także ze względu na to, że są niemal identyczne), a jeśli będziemy potrzebować podobnej notacji użyjemy notacji Wirtha.

2.3. Zależności kontekstowe

Gdzieś na pograniczu syntaksy i semantyki[4] leżą *zależności kontekstowe*. Omówienie ich osobno od składni wynika z tego, że opis składni języka

[4] Formalnie wypadałoby zaliczyć je do tej pierwszej, bo nie mówią one nic o znaczeniu elementów programu. Jednakże w praktyce programistycznej zależności kontekstowe badane są niezależnie od rozkładu strukturalnego (za dokonanie którego odpowiadają

za pomocą gramatyki bezkontekstowej jest bardzo prosty[5] i dlatego w powszechnym użyciu, jednak nie jest wystarczający. Spójrzmy na przykładowy tekst programu w języku C (Listing 2.1).

Listing 2.1. Przykładowy program

```
1 #include <stdio.h>

3 int nastepnik(int x) {
      return x+1;
5 }

7 int main(void) {
      printf("%d\n", nastepnik(41));
9     return 0;
  }
```

Oczywiście, program jest poprawny[6]. Jednakże zamiana w nim któregokolwiek (ale dokładnie jednego!) identyfikatora `x` na `y` (w linii 3 albo 4) powoduje błąd kompilacji, ale już zamiana `x` na `y` (lub na dowolny inny poprawny identyfikator[7]) w obu tych liniach — przywraca poprawność. Sprawa jest dla nas oczywista i nie ma w tym nic dziwnego.

Jednakże tej właściwości składniowej języka C[8] nie da się opisać za pomocą gramatyk bezkontekstowych[9]. Jest tak dlatego, że każda struktura składowa tego programu jest poprawna, a tylko poprawność strukturalna może być za pomocą gramatyk bezkontekstowych opisana — bez żadnego sprawdzania kontekstu. Natomiast linia 4 jest poprawna tylko i wyłącznie w kontekście jakiejś wcześniejszej deklaracji identyfikatora `x` i zamiana tej deklaracji na inną, równie poprawną deklarację identyfikatora `y` w linii 3 nie zepsuje owej struktury programu, natomiast sprawi, że jedna ze struktur

analizatory bezkontekstowe) tekstu programu, lecz w dalszym etapie, już na poziomie nadawania znaczenia poszczególnym elementom.

[5] Prosty zarówno dla człowieka projektującego język programowania, jak i dla programisty, który ma stworzyć interpreter lub kompilator tego języka — bo zależności bezkontekstowe sprawdza się (i rozkłada na składowe) względnie łatwo za pomocą względnie prostych, standardowych *parserów* (czyli programów przetwarzających tekst w danym języku formalnym w strukturę danych —najczęściej drzewiastą — odzwierciedlającą powiązania składniowe między elementami tego tekstu) dla gramatyk bezkontekstowych, automatycznie generowanych przez oprogramowanie takie, jak Yacc czy też GNU Bison [2, 41].

[6] I wyświetla odpowiedź na *Wielkie Pytanie o Życie, Wszechświat i Całą Resztę* [1]...

[7] Ba, można użyć tu nawet identyfikatora `nastepnik`!

[8] I bardzo wielu innych języków; właściwie prawie wszystkich powszechnie używanych języków programowania.

[9] Dowód formalny tego stwierdzenia zdecydowanie wykracza poza ramy tego skryptu. Mamy nadzieję jednak, że przedstawiona tu intuicja jest jasna.

(instrukcja w linii 4) nie będzie pasowała do całego kontekstu, w którym się ona znajduje.

Tak więc zależności kontekstowe są tym wszystkim w dokumentacji języka, które mówią o tym, jakich identyfikatorów możemy użyć w jakim miejscu (jak w powyższym przykładzie), w jaki sposób możemy odwoływać się do zadeklarowanych zmiennych różnych typów (w C++ bezkontekstowo poprawny jest napis `pies.szczekaj()`, ale całkowicie poprawny jest tylko w kontekście istnienia obiektu `pies`, na którym możemy wywołać bezargumentową metodę `szczekaj`), jakich operacji możemy dokonywać na jakich argumentach (`2+x` nie zawsze jest kontekstowo poprawne), ile argumentów przyjmuje dana funkcja (i jakich są one typów).

2.4. Semantyka

Semantyka języka programowania jest opisem znaczenia, jakie mają w programie poszczególne konstrukcje językowe. Różne są sposoby opisywania tego znaczenia — w sposób mniej lub bardziej formalny. Niektóre z nich [39] krótko tu przedstawimy.

Język potoczny ciągle jest najbardziej rozpowszechnionym (choć bardzo nieścisłym!) sposobem opisu znaczenia składników języka. O ile gramatyki bezkontekstowe wyśmienicie opisują strukturę programu i bardzo dobrze nadają się do automatycznej analizy składniowej — są więc w naturalny sposób chętnie przez projektantów języka wykorzystywane. Inaczej jest z semantyką, której formalne narzędzia (patrz niżej) wciąż pozostają w dużej mierze domeną informatyki teoretycznej.

Jak wygląda potoczny opis semantyki języka? Otóż po prostu dla każdej struktury opisanej produkcją gramatyki bezkontekstowej podajemy słownie jej znaczenie; na przykład dla produkcji:

```
InsWar = "if" "(" Wyr ")" Ins [ "else" Ins ] .
```

podajemy opis:

Wykonanie instrukcji warunkowej polega na zbadaniu wartości logicznej warunku, po czym wykonaniu pierwszej z podanych instrukcji, jeśli warunek był prawdziwy, lub też drugiej instrukcji, jeśli warunek był fałszywy i jednocześni fraza `else` *była obecna.*

My będziemy posługiwać się tutaj przy opisie języków i paradygmatów właśnie językiem potocznym.

Gramatyka atrybutywna także bywa w praktyce wykorzystywana. Polega ona na ustaleniu pewnych atrybutów, które mogą istnieć dla każdego symbolu gramatyki (tak terminalnego, jak i nieterminalnego), a wyrażają czynności reprezentowane przez dane struktury oraz wymagane zależności kontekstowe. Są one podczas parsowania (czyli analizy skła-

dniowej) zapamiętywane dla poszczególnych symboli, a do każdej produkcji podane są reguły sprawdzania atrybutów składowych i budowania wartości nowych atrybutów. Taka gramatyka jest używana wprost przy budowie kompilatorów czy interpreterów języków programowania.
Przykład dla definicji funkcji i jej wywołania poniżej (notacja z kropkami oznacza atrybuty danych symboli; podobnie jak zwykle stosowana w programowaniu obiektowym). Oczywiście, podobnie jak w poprzednich przykładach, są to tylko wybrane produkcje z całej definicji języka. Pod każdą produkcją zapisane są zależności między atrybutami symboli z danej produkcji.

```
DefFun = "def" Nazwa "(" ListaParForm ")" BlokIns .
```

$$\begin{aligned} f[\texttt{Nazwa}.wart].p &\leftarrow \texttt{ListaParForm}.lista \\ f[\texttt{Nazwa}.wart].k &\leftarrow \texttt{BlokIns}.kod \end{aligned}$$

```
WywFun = Nazwa "(" ListaParAkt ")" .
```

$$\texttt{WywFun}.kod \leftarrow \begin{cases} w(f[\texttt{Nazwa}.wart], \texttt{ListaParAkt}.lista) \\ \qquad \text{gdy } z(f[\texttt{Nazwa}.wart].p, \\ \qquad\quad \texttt{ListaParAkt}.lista), \\ błąd! \\ \qquad \text{w przeciwnym razie.} \end{cases}$$

Oczywiście wszystkie użyte powyżej symbole muszą być dobrze matematycznie zdefiniowane. Objaśnijmy je tutaj przynajmniej pokrótce.

— $lista, kod, wart$ są atrybutami symboli gramatycznych (różne symbole mogą mieć różny albo ten sam zestaw atrybutów, w zależności od potrzeb), przechowującymi odpowiednio: listę jakichś składowych (w naszym wypadku składowymi są nazwy parametrów formalnych lub wartości aktualnych, wraz z typami jednych i drugich), kod/pseudokod wygenerowany dla danej instrukcji oraz pewna prosta wartość (w tym przypadku identyfikator);
— f jest tablicą (asocjacyjną, patrz strona 61, Podrozdział 3.3.2.7) indeksowaną nazwami funkcji, która dla każdej napotkanej funkcji przechowuje listę jej parametrów formalnych (jako pole p) oraz kod tej funkcji (jako pole k);
— z jest funkcją, której wynikiem jest prawda logiczna, jeśli obie listy będące jej argumentami zgadzają się co do długości i typów swoich elementów, a fałsz w przeciwnym wypadku;
— w jest funkcją, której wynikiem jest kod wygenerowany dla wywołania funkcji, opakowany w odpowiednią jej *inicjalizację* (dotyczącą

zapewne głównie przekazania parametrów wejściowych, patrz strona 68, Podrozdział 4.1.2) i *finalizację* (związaną zwykle z oczyszczeniem stosu, zwróceniem wyniku i przekazaniem parametrów wyjściowych); patrz strona 69, Podrozdział 4.1.2);
— w końcu *błąd!* to bliżej tu nieokreślona reakcja na niezgodność parametrów wywołania (czyli aktualnych) z parametrami definicji (czyli formalnymi).

Jak zinterpretować teraz te zapisy? Pierwsza reguła mówi o tym, że po napotkaniu definicji funkcji należy kod odpowiadający ciału tej funkcji oraz jej parametry formalne gdzieś zapamiętać. Druga reguła mówi o tym, że przy wywołaniu należy sprawdzić zgodność parametrów aktualnych z zapamiętaną listą parametrów formalnych i jeżeli są w zgodzie, to wygenerować kod wywołania funkcji wraz z jego odpowiednim opakowaniem.

Formalne semantyki teoretyczne ze względu na ich użycie w informatyce teoretycznej zasługują także na wspomnienie, choć bliżej ich omawiać nie będziemy. Do tych semantyk należą między innymi:
— *semantyka denotacyjna*, która znaczenie każdego zapisu w analizowanym języku opisuje przez odpowiednią funkcję matematyczną działającą na stanie maszyny; odpowiada to z grubsza *kompilacji*, czyli tłumaczeniu jednego języka programowania na inny (w tym wypadku matematyczny);
— *semantyka operacyjna* opisująca każdą strukturę języka przez czynności, które jej odpowiadają — zwykle na jakiejś *maszynie abstrakcyjnej* (jak maszyna RAM, maszyna SECD, maszyna Turinga[10] i inne); to z kolei odpowiada mniej więcej *interpretacji*, czyli wykonywaniu na bieżąco danych zapisów języka;
— *semantyka aksjomatyczna* traktuje frazy danego języka jako obiekty matematyczne i podaje ich znaczenie przez aksjomaty logiczne, które muszą przez dane frazy być spełnione; dzięki takiemu podejściu dowodzenie poprawności programów staje się względnie proste i przykładem takiej semantyki jest wspominana już wcześniej *logika Hoare'a*.

2.5. Pytania i zadania

1. Wyjaśnij pojęcia: język, składnia, semantyka, token, leksem, symbol terminalny, symbol nieterminalny, produkcja, zależność kontekstowa.
2. Z jakich dwóch elementów składa się opis każdego języka?

[10] Choć maszyna Turinga rzadko jest tu wykorzystywana ze względu na jej wysoki stopień abstrakcji.

3. Czy $AB = BA$ dla każdych dwóch zbiorów napisów A, B? Kiedy tak jest, a kiedy nie?
4. Co oznacza $\{\texttt{a}, \texttt{bc}\}^*$?
5. Co oznacza ε?
6. Napisz wyrażenie regularne opisujące zapis liczby zmiennoprzecinkowej w języku C.
7. * Napisz wyrażenie regularne opisujące zapis liczby zespolonej w Pythonie.
8. Rozważmy zapis produkcji w notacji Wirtha:

```
ListaZmiennych
    =  [  Zmienna  {  ","  Zmienna  }  ]  .
```

   Zamień ten zapis na tradycyjny matematyczny, bez skrótów.
9. * Rozważmy zapis produkcji w notacji Wirtha:

```
ListaList
    =  Wartość  {  ","  Wartość  }
       {  ";"  Wartość  {  ","  Wartość  }  }  .
```

   Zamień ten zapis na tradycyjny matematyczny, bez skrótów.
10. Znajdź wyprowadzenie dla przykładowej instrukcji podanej na stronie 22 w Przykładzie 2.13, zakładając, że istnieją podane tam potrzebne wyprowadzenia.
11. Podaj kilka przykładów zależności kontekstowych w znanych Ci językach programowania.
12. * Zdefiniuj składnię notacji Wirtha za pomocą notacji Wirtha.
13. Zdefiniuj składnię instrukcji pętli `for` oraz `while` w języku C za pomocą notacji Wirtha. Nie wgłębiaj się w kolejne poziomy definicji (jak wyrażenia, instrukcje itd.).
14. * Spróbuj opisać semantykę instrukcji z Przykładu 2.13 (przy założeniu, że jest taka, jak w C) za pomocą gramatyki atrybutywnej.
15. Rozważmy następujące produkcje (zapisane w notacji Wirtha, `S` jest symbolem startowym):

```
A = "a" .
B = "bb" .
S = AA | AB | SS .
```

   Które z następujących słów: `a`, `abbaa`, `abba`, `S`, `aaaa`, `A` należą do języka generowanego przez tę gramatykę? Przedstaw dla nich odpowiednie wyprowadzenia.

Rozdział 3

Dane i typy danych

3.1. Wiązania

Zanim zaczniemy bardziej szczegółowo omawiać atrybuty danych (a później innych obiektów), musimy wprowadzić pojęcie *wiązania* [37, 54]. Wiązanie polega na połączeniu (w sensie logicznym) pewnych bytów — w naszym przypadku chodzi o ich połączenie w języku programowania czy też w programie. Wiązanie może dotyczyć na przykład zmiennej i jej wartości, czy też obiektu z odpowiednią metodą wirtualną.

Równie ważny jak samo pojęcie wiązania jest czas, kiedy ono następuje. W tym kontekście można rozumieć moment wiązania jako czas decyzji czyli dania odpowiedzi na pewne pytanie istotne dla działania języka/programu (na przykład: jaką wartość ma dana zmienna? który konkretny podprogram oznacza dana metoda wirtualna?). Kiedy może dokonać się wiązanie?

W czasie projektowania języka. Projektanci języka dokonują wyborów pewnych słów kluczowych, które wiążą z pewnym znaczeniem dokumentując język. Dotyczy to konstrukcji algorytmicznych, podstawowych typów, operatorów itp.

Przykład: wybór słowa `while` jako słowa kluczowego oznaczającego pętlę dopóki, a symbolu `=` jako oznaczenia podstawienia.

W czasie implementowania języka. Teraz muszą dokonać się wybory, które dokumentacja języka pozostawia otwarte — jak na przykład sposób reprezentacji pewnych danych w pamięci, zakres liczb całkowitych i zmiennoprzecinkowych (co może zależeć od maszyny), obsługa pewnych błędów wykonania (jak dzielenie przez zero)...

Przykład: wybór zakresu i reprezentacji maszynowej typu `int`; wybór reprezentacji maszynowej dla konkretnych literałów, na przykład `14`, `"Ala ma kota."`.

W czasie pisania programu. W tym momencie programista używający języka wybiera nazwy zmiennych, ich typy (jeśli są deklarowane) i inne elementy pisanego programu.

Przykład: wybór nazwy `RysujKwadrat` dla pewnego podprogramu.

W czasie kompilacji programu. Kompilator wiąże elementy programu z odpowiednią ich reprezentacją w kodzie maszynowym[1].

Przykład: związanie zmiennych statycznych z adresami wirtualnymi; związanie zmiennych z zadeklarowanym typem.

W czasie konsolidacji programu. Wiele kompilatorów stosuje *kompilację rozdzielną*, co oznacza, że można niezależnie kompilować różne moduły, biblioteki itp. Jednak przed uruchomieniem programu należy go z używanymi modułami połączyć, czyli *skonsolidować*. Konsolidator[2]

[1] Lub ogólniej: w języku docelowym, bo kompilacja nie musi dawać wyniku w kodzie maszynowym.

[2] Często potocznie: linker (z ang.); stąd też częste 'linkować' zamiast 'konsolidować'.

wiąże wszystkie elementy, które muszą być powiązane, by program mógł się uruchomić.

Przykład: związanie wywołania funkcji `RysujKwadrat` z jej implementacją w innym module.

W czasie ładowania programu. System operacyjny ładując program do pamięci (i przygotowując go do uruchomienia) musi powiązać przede wszystkim adresy wirtualne (które w kodzie wykonywalnym pozostawia konsolidator) z realnymi adresami fizycznymi, w których program się znajdzie.

Przykład: związanie zmiennych statycznych z adresami fizycznymi (za pośrednictwem adresów wirtualnych).

W czasie działania programu. W końcu bardzo wiele wiązań dokonuje się już w czasie działania programu.

Przykład: zmiana wartości zmiennej przez podstawienie; zmiana adresu zmiennej dynamicznej przez jej realokację.

Wiązania dzieli się też na *statyczne* (takie, które dokonywane jest **przed rozpoczęciem programu** i w dodatku nie zmieniają się w czasie jego działania; w powyższym wyliczeniu są to wszystkie etapy do ładowania programu włącznie — poza ostatnim punktem) i *dynamiczne* (czyli te, które dokonują się lub zmieniają **podczas pracy programu**).

Moment dokonywania różnych wiązań ma bardzo duży wpływ na charakter języka. Dokonanie każdego wiązania zabiera jakiś czas, stąd, jeśli zależy nam na efektywności kodu wynikowego należy wybrać język oferujący jak najwięcej wczesnych wiązań (a więc statycznych). Takie języki zwykle mogą być kompilowane, a kompilacja ma to do siebie, że stara się możliwie dużo rzeczy powiązać statycznie — dlatego też zwykle języki kompilowane są efektywniejsze, jeśli chodzi o czas wykonywania.

Języki takie jednak są zwykle mniej elastyczne i więcej pracy wymagają od programisty (są więc językami niższego poziomu i są mniej efektywne, jeśli chodzi o wkład pracy człowieka). Bowiem to wiązania dynamiczne elastyczność zapewniają. Wiele języków — z powodu takiej a nie innej specyfikacji — nie nadaje się do kompilacji, bo wiele decyzji musi być odłożonych do momentu wykonania programu (bo na przykład zbyt wiele zależy od danych podanych programowi). Języki takie muszą być *interpretowane* (lub częściowo kompilowane, częściowo interpretowane), a interpreter wszelkie decyzje (w tym o bardzo wielu wiązaniach) podejmuje dopiero w czasie wykonywania programu.

Typowym, a prostym przykładem tej sprzeczności interesów jest implementacja tablic (o czym więcej w Podrozdziale 3.3.2.3 na stronie 50). Jeśli język nakazuje wszystkie tablice deklarować z podaniem ich dokładnych wymiarów i typów elementów, to można wszystkie atrybuty tablicy (poza oczywiście przechowywanymi wartościami, jeśli nie jest to tablica stała)

związać przed wykonaniem programu — i taki język będzie w operacjach na tablicach bardzo szybki. Jednakże, deklarując w programie tablicę o 100 elementach narażamy się na marnotrawstwo (jeśli zwykle używamy tylko 10–20 elementów), albo na ograniczenie działania programu — bo 101 elementów już się nie zmieści.

Z drugiej strony, tablice z rozmiarem dynamicznym stwarzają problemy wydajnościowe, bo alokacja pamięci (a to dość kosztowna operacja) musi być odłożona do czasu uruchomienia programu, adresy poszczególnych komórek też są wyliczane w czasie pracy programu, bo nie mogą być wyliczone z góry, a jeśli dodatkowo język dopuszcza zmianę rozmiaru tablicy, musi być przewidziana możliwość jej realokacji. Jednak dla programisty wygodna w użyciu takich tablic jest wyraźna.

3.2. Pojęcie danej

Omawianie poszczególnych aspektów programowania zaczniemy od pojęcia wspólnego dla wszystkich chyba paradygmatów, to jest od *danej*.

Dane są abstrakcjami pamięci, to jest udostępniają w jakiś sposób użytkownikowi języka (czyli programiście) dostęp do pamięci, ale z pewnymi udogodnieniami. Im więcej udogodnień, im bardziej pamięć jest obudowana i ukryta (w swej surowej postaci) przed programistą, tym język, z jakim mamy do czynienia, jest *wyższego poziomu* (jak Haskell, Prolog, Python, Java). I odwrotnie — jeśli mamy bezpośredni dostęp do pamięci w prosty sposób, a do tego jeszcze mało kontrolowany, mówimy o *językach niskiego poziomu* (jak C, asemblery). Oczywiście, podobną analizę można prowadzić także dla innych elementów języka (jak typy, podprogramy) i ich abstrakcyjność także decyduje o usytuowaniu języka w odpowiednim miejscu skali „wysoki–niski".

Dana sama w sobie jest pojęciem abstrakcyjnym i należałoby ją zdefiniować matematycznie — jako parę uporządkowaną złożoną z identyfikatora danej (w zależności od podejścia może to być adres w pamięci, nazwa danej w programie, a może coś jeszcze), który się nie zmienia; oraz wartości danej, która zmienić się może podczas wykonywania programu (ale nie we wszystkich językach programowania) [61, 68]. Bardziej jednak od ścisłej definicji obchodzić nas będzie zestaw atrybutów jakie ma dana (często jest to zmienna, ale nie jest to konieczne) [37, 52, 55]. Możemy tu wyróżnić:

— *adres*, to jest miejsce (miejsca) w pamięci (być może zewnętrznej), które każda dana musi zajmować;
— *nazwa*, która identyfikuje jakoś daną w programie;
— *wartość*;

— *typ*, to jest zbiór wartości, które może dana przyjmować (wraz z operacjami, jakie można na niej wykonywać);
— *okres życia*, czyli czas (wykonywania programu) w którym dana istnieje;
— *zakres widoczności*, czyli miejsce w programie, gdzie do danej można się odwołać.

3.2.1. Nazwa i adres

Nie każda dana musi mieć nazwę, a niektóre mają nazwy złożone — `t[10]`, `student.nazwisko`. Z drugiej strony, adres musi mieć każda dana, choć ten adres nie musi być wprost dostępny (szczególnie w językach wyższego poziomu). Pewne zależności (a właściwie „niezależności") między daną, jej nazwą i jej adresem przedstawiają Listingi 3.1–3.3 (fragmenty programów w języku C).

Listing 3.1. Różne zmienne o tej samej nazwie

```
...
int f(void) {
  int x;
  ...
}
...
int g(void) {
  float x;
  ...
}
...
```

W Listingu 3.1 widzimy **dwie zmienne o tej samej nazwie** `x`. Czasy życia tych zmiennych na pierwszy rzut oka nie zachodzą na siebie, ale może być tak, że funkcja `g` wywołuje funkcję `f` (lub odwrotnie) i wtedy mamy sytuację, że istnieją **dwie zmienne o tej samej nazwie w tym samym momencie** — oczywiście dostęp jest tylko do jednej z nich, bo jedna przesłania drugą (a w tym przykładzie obie są lokalne, więc i bez przesłaniania nie są widoczne).

Listing 3.2. Jedna zmienna o różnych (być może) adresach

```
...
int f(void) {
  int x;
  ...
}
...
int g(void) {
  ...
```

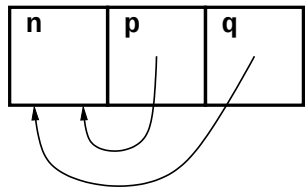
```
    y = f();
    ...
    z = f();
    ...
}
...
```

Listing 3.2 pokazuje sytuację inną, gdy jedna zmienna (`x`) ma nieciągły okres życia, bo jest tworzona i niszczona dwa razy przy wywołaniu funkcji `f` — **może wtedy mieć dwa różne adresy**, choć w logice programu jest to ta sama zmienna. Sytuacja mogłaby być bardziej skomplikowana, gdyby funkcja `f` była rekurencyjna — wtedy jedna (logicznie i „zapisowo") zmienna miałaby w danym momencie kilka wcieleń, a więc kilka adresów (bo każde wywołanie funkcji tworzy swoje zmienne lokalne, a w rekurencyjnym wywołaniu kolejne zmienne tworzone są zanim poprzednie zostaną zniszczone)! Widoczna oczywiście znowu byłaby tylko jedna z nich, ta „najgłębsza", w całej rekurencji utworzona ostatnio.

Listing 3.3. Aliasowanie

```
...
int n;
int *p;
int *q;
...
p = &n;
q = &n;
...
```

W końcu Listing 3.3 przedstawia *aliasowanie* zmiennych, co jest sytuacją w pewnym sensie odwrotną do poprzednich. Polega ono na tym, że pewne dane są widoczne pod różnymi nazwami — a więc **różne nazwy są związane z tym samym adresem** i — co więcej — są one jednocześnie dostępne. Tutaj przez trzy nazwy (`n`, `*p`, `*q`) możemy odnosić się w tym samym momencie do tej samej danej (Rysunek 3.1 przedstawia tę sytuację; dane są ustawione sekwencyjnie nieprzypadkowo, bowiem najprawdopodobniej taki będzie ich układ w pamięci operacyjnej). Aliasowanie zwykle nie jest pożądaną sytuacją i może prowadzić do zagmatwania kodu. Jednakże pewne języki nie unikają go, a wręcz przeciwnie — wykorzystują we wbudowanych mechanizmach. Więcej o aliasowaniu powiemy trochę przy okazji omawiania elementów języków funkcyjnych (Podrozdział 5.6.2.1, strona 99) oraz języka Python (Podrozdział 7.3.2, strona 146), gdzie jest ono jedną z ważnych cech języka.



Rysunek 3.1. Sytuacja po wykonaniu siódmego wiersza z Listingu 3.3

3.2.2. Okres życia danej

Ważnym atrybutem każdej danej jest jej *okres życia* (także: *czas życia*). Jest to okres podczas wykonywania programu, kiedy dana jest związana z miejscem w pamięci — to jest kiedy istnieje. Innymi słowy, jest to **czas od *alokacji*[3] pamięci na tę daną do *dealokacji*[4] tej pamięci**.

Można też mówić o okresie życia wiązania, który trwa wtedy, kiedy dane wiązanie istnieje — jest to czas od momentu zaistnienia wiązania do jego zniszczenia.

Okres życia danej nie pokrywa się zwykle z okresem życia wiązania tej danej z jej nazwą. Dana może istnieć, ale nie być dowiązana do nazwy, lub też być związana z różnymi nazwami w różnym momencie — ogólnie okres życia wiązania danej z nazwą zawiera się w okresie życia tej danej. Przykładem takiej danej jest tablica alokowana w wierszu Listingu 3.4 (język C). Program zawierający taki kod jest niedobry (patrz też o zgubionych danych, na stronie 55, w Podrozdziale 3.3.2.5), bowiem po drugiej alokacji, w wierszu 5, poprzednio alokowana tablica nadal istnieje w pamięci, ale nie ma już do niej przywiązanej żadnej nazwy. Zostanie więc tam do końca działania programu, zajmując niepotrzebnie pamięć (bo C nie używa zbierania nieużytków, patrz strona 56, Podrozdział 3.3.2.5).

Listing 3.4. Dana niezwiązana z nazwą

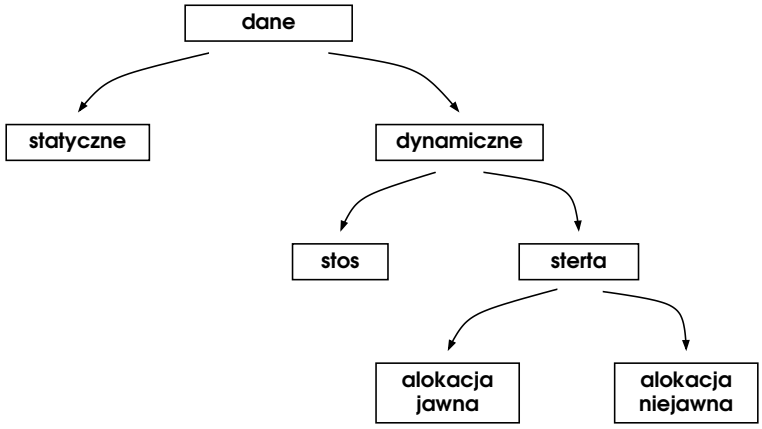
```
  ...
2 int *tab;
  ...
4 tab = calloc(10, sizeof(int));
  tab = calloc(5, sizeof(int));
6 ...
```

[3] Alokacja (inaczej: przydział) pamięci to zarezerwowanie pewnego obszaru wolnej pamięci dostępnej dla programu na przechowywanie danej.

[4] Dealokacja (inaczej: zwolnienie) pamięci to „zwrócenie" zaalokowanego wcześniej obszaru do puli pamięci wolnej.

3.2.2.1. Podział danych ze względu na okres życia i miejsce alokacji

Schemat z Rysunku 3.2 pokazuje podział danych ze względu na czas ich życia, ich miejsce w pamięci oraz sposób alokacji.



Rysunek 3.2. Rodzaje danych

Dane statyczne. Są to takie dane, które mają pamięć związaną ze sobą statycznie, czyli przed uruchomieniem programu a adres tej pamięci się nie zmienia przez cały czas działania programu[5]. Pamięć na takie dane przydzielana jest wirtualnie w czasie kompilacji programu, a fizycznie w czasie jego ładowania. Pamięć ta może znajdować się — w zależności od wymagań i możliwości sprzętu oraz od sytuacji — w osobnym segmencie danych lub bezpośrednio w kodzie wykonywalnym. W ten sposób traktowane są zwykle wszelkie stałe programu, zmienne globalne, zmienne z deklaratorem `static` w języku C (ale nie w językach obiektowych pochodnych od C!). W języku Fortran od samego jego początku (czyli od lat 50.) do lat 80. tak były traktowane także zmienne lokalne funkcji — co uniemożliwiało korzystanie w Fortranie z rekurencji, ale (wraz z innymi cechami) czyniło go jednym z najbardziej efektywnych (do dziś!) języków programowania obliczeń, bo zmienne statyczne mogą być w użyciu nawet wielokrotnie szybsze od dynamicznych.

Dane dynamiczne. Są to wszelkie dane niestatyczne, a więc takie, dla których pamięć jest alokowana (lub zmienia się) dopiero w czasie pracy programu. W zależności od miejsca w pamięci możemy je dalej podzielić.

[5] Oczywiście wartości zmieniać się mogą!

Dane dynamiczne alokowane na stosie. Są one zawsze alokowane i zwalniane niejawnie (poza wyjątkowym wypadkiem asemblerów), czyli poza udziałem programisty, i korzystają z pamięci stosu maszynowego (patrz też strona 4, Podrozdział 3). Są one niezbędne „jedynie”[6] wtedy, gdy język pozwala na rekurencyjne definicje podprogramów. W takim bowiem przypadku nie ma górnego ograniczenia na liczbę niezakończonych wywołań jakiejkolwiek funkcji[7]. Z pomocą przychodzi właśnie stos, na który można wkładać dane i z którego można je zdejmować (w odwrotnej kolejności niż są wkładane). Kolejność stosowa odpowiada dokładnie kolejności w jakiej mają być obsługiwane zagnieżdżone podprogramy — ten który wcześniej się zaczął musi być później zakończony. (więcej na ten temat w Rozdziale 4, tutaj tylko Rysunek 3.3 przedstawiający stos podczas wywoływania podprogramów z Listingu 3.5).

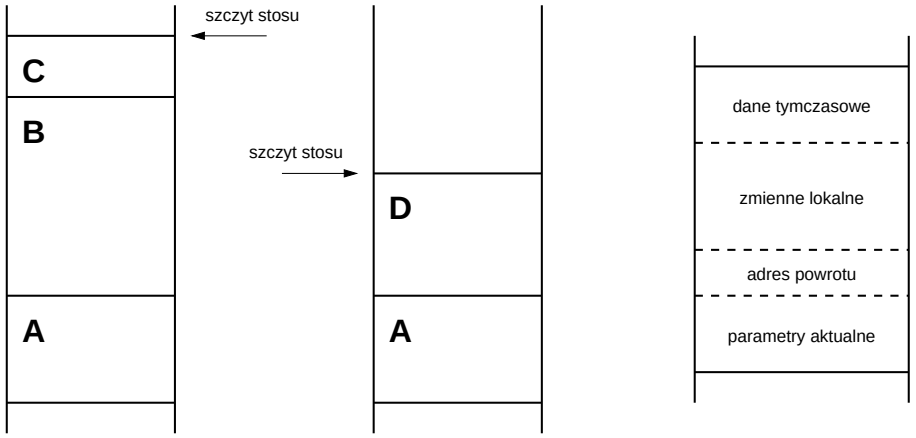
Listing 3.5. Przykład zagnieżdżonego wywoływania podprogramów

```
...
void A(...){
   ...
   B(...);
   ...
   D(...);
   ...
}
...
void B(...){
   ...
   C(...);
   ...
}
...
void C(...){
   ...
}
...
void D(...){
   ...
}
...
```

Tak więc zmienne lokalne podprogramów (a w niektórych językach także zmienne lokalne bloków — jak może to być w C++) są alokowane na stosie. Mają one wiele zalet:

[6] Cudzysłów, bo współcześnie prawie żaden język — poza niektórymi językami najniższego poziomu — nie wyklucza rekurencji.

[7] Oczywiście poza dostępnym maksymalnym rozmiarem stosu.



Rysunek 3.3. Stan stosu maszynowego w czasie wywołania funkcji `A`: po dotarciu do linii 17 (po lewej); po dotarciu do linii 21 (w środku); przykładowy (bo zależy od implementacji) powiększony obszar (*ramka wywołania*) jednego z podprogramów (po prawej)

— są tylko wtedy alokowane, gdy są potrzebne (a więc na czas działania podprogramu), a po skończeniu podprogramu pamięć jest dealokowana;
— przydzielanie i zwalnianie pamięci jest automatyczne;
— wszystko poza samym miejscem w pamięci (więc typ, rozmiar itp.) może być związane statycznie, co zwiększa efektywność;
— pozwalają na rekurencję.

Ich wady to:

— pewien narzut czasowy (w stosunku do zmiennych statycznych) związany z alokacją i dealokacją oraz z adresowaniem pośrednim (adres jest wyliczany względem aktualnego szczytu stosu, nie jest statyczny);
— w pewnych sytuacjach stos nie jest (rozmiarowo) wystarczający na przechowywanie dużych strukturalnych zmiennych lokalnych — wtedy na stos trafia tylko ich adres, a pamięć na właściwe dane jest alokowana niejawnie na stercie (patrz niżej);
— nie są dostatecznie elastyczne — poza zmiennymi lokalnymi (i parametrami) podprogramów nie mają zastosowań.

Dane dynamiczne alokowane na stercie. Sterta to obszar pamięci dostępnej dla procesu, usytuowany poza kodem, danymi i stosem. Może on być dowolnie zarządzany przez proces i służy zwykle do przechowywania danych dynamicznie tworzonych. Podzielony jest zwykle na jakieś bloki, z których niektóre są wolne (na początku wszystkie), a inne dynamicznie, w miarę potrzeb przydzielane danym. Sterta jest najbardziej elastycznym miejscem

alokacji pamięci, ale z drugiej strony najwolniejszym. Alokacja pamięci na stercie może odbywać się na dwa sposoby.

Dane dynamiczne alokowane na stercie jawnie. Alokacji jawnej musi dokonać sam programista: w C służą do tego funkcje `malloc` i `calloc`, w C++ — instrukcja `new`, a w Pascalu — procedury `New` oraz `GetMem`. Dane takie mogą być dealokowane jawnie (`free`, `delete`, `Dispose`, `FreeMem`), mogą też być dealokowane niejawnie jeśli istnieje odpowiedni mechanizm języka (Podrozdział 3.3.2.5 na stronie 56) lub zostaną oprogramowane odpowiednie destruktory, które robią to automatycznie. Takie dane nie mają nazwy prostej, lecz dostępne są przez wskaźnik/referencję (jak na przykład `*p` w C, czy też `p^` w Pascalu).

Zmienne tego rodzaju mają następujące zalety:
— są maksymalnie elastyczne dla programisty;
— nie wymagają (jak stos) wykonywania dealokacji w określonym porządku;
— pozwalają na statyczną implementację wiązania typu, co umożliwia ich efektywne stosowanie w językach kompilowanych;
— pozwalają na ręczne tworzenie niemalże dowolnych struktur danych (listy, drzewa, grafy, struktury potencjalnie nieskończone);
— rozmiar alokowanej pamięci jest ograniczony zwykle tylko przez pamięć przydzieloną procesowi (która może być znacznie większa niż pamięć stosu) i ewentualną fragmentację zewnętrzną sterty.

Są takie dane jednak najbardziej kłopotliwe w użyciu:
— alokacja pamięci na stercie zajmuje dużo czasu w związku z zarządzaniem stertą;
— także odwołania do tych zmiennych są wolniejsze, bo wyliczanie adresów musi być w pełni dynamiczne i będzie przez to wolniejsze niż danych stosowych;
— programista musi być świadomy tego co robi, bardzo łatwo tutaj o przeoczenia, które mogą doprowadzić do wadliwego działania programu (patrz strona 3.3.2.5).

Dane dynamiczne alokowane na stercie niejawnie. Najwygodniejszymi dla programisty danymi są te alokowane na stercie, ale obsługiwane przez język automatycznie. Takie dane są alokowane w miarę potrzeb i dealokowane, gdy są niepotrzebne, niejako za plecami programisty, przy okazji innych czynności. Takie rozwiązania oferują języki najwyższego poziomu (jak Perl, Python, PHP, Lisp, Scheme, Haskell, Prolog), w których istnieją skuteczne mechanizmy zbierania nieużytków (Podrozdział 3.3.2.5). Są to dane zapewniające programiście najbardziej wygodną pracę, aczkolwiek — tak, jak inne rozwiązania odkładające wiązania wielu atrybutów do czasu

uruchomienia programu — okupione jest to dodatkowym kosztem czasowym i pamięciowym.

3.2.3. Zakres widoczności

Dane — a ściślej mówiąc ich nazwy — są dostępne lub nie w różnych miejscach programu. Obszar tekstu programu (niekoniecznie ciągły), w którym dana jest pod swoją nazwą widoczna nazywamy *zakresem widoczności* tej danej. Ważne jest, by należycie odróżnić zakres widoczności od okresu życia — **zakres widoczności to pojęcie przestrzenne i odnosi się do obszaru kodu programu, natomiast okres życia jest pojęciem czasowym i odnosi się do czasu działania programu**. Do zakresu widoczności odnoszą się takie pojęcia jak: nazwa lokalna względem bloku (taka, która jest dostępna w tym bloku), nazwa nielokalna względem bloku (taka, która jest dostępna w bieżącym bloku, ale zadeklarowana w innym), nazwa globalna (czyli widoczna w całym programie, o ile nie jest przesłonięta).

Zakres widoczności — podobnie jak inne atrybuty — także może być statyczny lub dynamiczny. Bez względu na to możliwe jednak jest *przesłanianie* nazw, to znaczy: z kilku takich samych nazw widoczna jest ta, która jest „bliżej" (w sensie poprzednika statycznego lub dynamicznego — patrz niżej) odwołania do niej.

Statyczny zakres widoczności. Wiele współczesnych języków (w tym wszystkie kompilowane) posługuje się statycznym zakresem widoczności. Może on — jak sama nazwa wskazuje — być określony przed uruchomieniem programu na podstawie samego tekstu programu. Ustalany jest on w ten sposób, że w chwili napotkania odwołania do jakiejś nazwy kompilator sprawdza, czy jest ta nazwa zadeklarowana w bieżącym bloku; jeśli nie, to sprawdzany jest blok **zawierający** blok badany (zwany *poprzednikiem statycznym*); jeśli i tu nie ma deklaracji, sprawdzanie na tej zasadzie powtarza się do skutku (lub do stwierdzenia błędu).

Rozważmy Listing 3.6 przedstawiający fragment programu napisanego w hipotetycznym języku (składnia analogiczna do C, przy czym `print` po prostu wyświetla swoje argumenty). W przypadku statycznego zakresu widoczności program wyświetli wartość `1`, bo w czasie kompilacji nazwa `x` występująca w linii 3 związana jest z deklaracją w linii 1 (poprzednikiem statycznym funkcji `a` jest obszar globalny programu).

Zakres statyczny jest oczywiście szybszy (bo ustalany przed uruchomieniem programu) i oferuje większy „porządek" w programie, przez sprawdzenie poprawności odwołań do nazw przed jego uruchomieniem. Z drugiej strony, oczywiście, ogranicza elastyczność rozwiązań.

Listing 3.6. Zagnieżdżone definicje funkcji i zakres widoczności nazw

```
1  int x = 1;
   void a(void) {
3    print(x);
   }
5  void b(void) {
     int x = 2;
7    a();
   }
9  void main(void) {
     b();
11 }
```

Dynamiczny zakres widoczności. Zakres może być jednak ustalany inaczej — to jest dynamicznie, czyli w czasie działania programu[8] — tak jak na przykład w Pythonie. Ustalanie zakresu dynamicznego odbywa się bardzo podobnie do ustalania zakresu statycznego, lecz używa się pojęcia *poprzednika dynamicznego* zamiast statycznego — to znaczy, że jeśli nie znajdzie się deklaracji nazwy w bloku, szuka się w bloku **wywołującym** dany blok, a nie zawierającym go.

Gdyby w Listingu 3.6 używać zakresu dynamicznego, to wyświetlona zostałaby wartość `2` – bo poprzednikiem dynamicznym funkcji `a` w tym programie jest zawsze[9] funkcja `b`.

Zakres dynamiczny jest wolniejszy od statycznego, bo odłożony do czasu uruchomienia programu; ponadto inne sprawdzenia (jak zgodność typów) muszą też być odłożone do uruchomienia programu; no i w końcu nie sprzyja zachowaniu czytelności kodu i przejrzystości odwołań. Za to jest rozwiązaniem bardziej elastycznym.

Z zakresem widoczności związane jest także pojęcie środowiska referencyjnego (Podrozdział 4.1.3.4 na stronie 74).

3.3. Typy danych

Typ — od strony teoretycznej — jest zbiorem dopuszczalnych wartości jakie dana może przyjmować, wyznacza też zbiór operacji dopuszczalnych

[8] Właściwie to zakres statyczny też może być ustalany dopiero w czasie działania programu i niektóre języki interpretowane, ale z zakresem statycznym tak właśnie czynią. Stąd też czasem używa się nazwy *zakres leksykalny* zamiast statyczny. Z drugiej strony zakres dynamiczny musi być ustalany na bieżąco, w czasie działania programu, więc tu innej nazwy nie ma.

[9] Bo w różnych momentach działania programu (albo też w różnych uruchomieniach) poprzedniki dynamiczne dowolnej funkcji mogą być różne (bo może ona być wywoływana z różnych miejsc programu) — inaczej niż poprzednik statyczny, który zawsze jest jeden.

na danej. Technicznie, typ danej ustala także jej reprezentację w pamięci komputera oraz traktowanie przez kompilator/interpreter.

Po co są typy w językach programowania? Wszystko jest w końcu i tak reprezentowane jako ciąg bitów i wszelkie operacje przetwarzają tylko jedne ciągi bitów na drugie... Oczywiście, typy oferują nam kolejne (po danych) narzędzie abstrakcji (czyli „odgrodzenia" się od maszynowych, bitowych wnętrzności maszyny) i interpretacji owych bitów. Ściśle mówiąc, mamy do czynienia z kilkoma aspektami wykorzystania typów:

— *zgodność typów* pozwala kompilatorowi/interpreterowi kontrolować, czy odpowiednie dane są użyte w odpowiednim kontekście, nie pozwalając na operacje niezdefiniowane w języku (jak dodawanie znaku do liczby[10]);

— *przeciążanie operatorów, funkcji, metod* pozwala kompilatorowi/interpreterowi dobierać operacje w zależności od kontekstu — inaczej bowiem rozumiany jest znak `+` w każdym z następujących wyrażeń w Basicu: `2+3`, `2.0+3.0`, `"2"+"3"` — pierwszy oznacza dodawanie całkowite, drugi dodawanie zmiennoprzecinkowe, a trzeci sklejanie napisów[11]; analogiczna sytuacja będzie przy wywołaniu funkcji `len(s)` w Pythonie (interpretacja długości obiektu `s` zależy od jego typu), czy też metody `x.rysuj()` w C++, C#, Javie, Pythonie o ile obiekt `x` należy do jakiejkolwiek klasy posiadającej metodę `rysuj`;

— w końcu *typy abstrakcyjne*[12] pozwalają na modularyzację programów — czyli podział na niezależne implementacyjnie części, a więc ułatwiają testowanie, znajdowanie błędów i pracę zespołową.

Każda dana musi być przed swoim użyciem oczywiście związana z typem i wiązanie to może dokonywać się na różne sposoby. Pierwszy podział to oczywiście moment dokonania wiązania. Może się to dokonać przed uruchomieniem programu (zwykle w trakcie kompilacji) i mamy do czynienia wtedy ze statycznym wiązaniem typów. Może to też być odłożone do czasu uruchomienia programu (zwykle przez interpreter) i wtedy jest to wiązanie statyczne.

Inna kwestia, to deklarowanie typów zmiennych: język może wymagać takich deklaracji (Pascal, C, C++, Java, C#) albo może ich nie wymagać. Języki, które nie wymagają takich deklaracji mogą stosować wnioskowanie o typie na podstawie kontekstu (Haskell[13], ML, Miranda), na podsta-

[10] Ciekawe jak różne języki rozwiązują ten konkretny problem dodawania (`+`) znaku i liczby: na przykład Python, Pascal, Haskell, Ada nie pozwalają na to (bez jawnej konwersji), natomiast C (i C++) oraz PHP pozwalają — jednak wyrażenie `2+'3'` da w nich całkiem różne wyniki...

[11] Wniosek z tego, że nawet języki najbardziej prymitywne (poza asemblerami) mają wbudowane operatory przeciążone, bo inaczej jest maszynowo wykonywana każda z trzech wymienionych tu operacji!

[12] Klasy abstrakcyjne to pojęcie węższe, o czym dalej w Podrozdziale 4.2.

[13] Więcej o wnioskowaniu o typie w Haskellu w Rozdziale 5.

wie przypisania/konstruktora (Python, PHP, Bash), na podstawie nazwy zmiennej (Fortran[14], Perl[15], stare wersje języka BASIC[16]).

I znowu bardzo ważną sprawą jest ów moment dokonania wiązania nazwy/danej z typem. Mamy więc języki ze statycznym wiązaniem typów (w których typy danych są ustalane przed uruchomieniem programu) i dynamicznym (w których typy danych są ustalane na bieżąco, podczas wykonywania programu).

Ponownie widzimy tutaj konflikt między uporządkowaniem i szybkością wykonania programu (które daje wiązanie statyczne) a swobodą, elastycznością i uniwersalnością rozwiązania (które daje wiązanie dynamiczne). Dynamiczne wiązanie typów można jednak zaimplementować w kompilatorach, o ile każda dana oprócz swej wartości przechowuje deskryptor typu. Wtedy oczywiście kompilator może wygenerować kod, który będzie przy każdej okazji sprawdzał, z jakim typem ma do czynienia i podejmował odpowiednie działania (sprawdzanie zgodności, automatyczne konwersje, wywołanie odpowiednich metod). Oczywiście taki kod jest większy, działa powolniej i kompilacja może się po prostu nie opłacać w takim wypadku — szczególnie, gdy mamy język z dużą ilością różnych rodzajów typów danych, w szczególności skomplikowanych strukturalnych.

3.3.1. Zgodność typów

Jak wspomnieliśmy wcześniej, typ wyznacza też operacje, jakie mogą być na jego wartościach wykonywane — a więc *zgodność typów* polega na tym, że typy danych użytych w różnych miejscach zgadzają się z typami, które są tam spodziewane. Jakie może być sprawdzanie zgodności typów?

Statyczne/dynamiczne. Sprawdzanie zgodności typów musi odbywać się oczywiście po wiązaniu typów i może być statyczne — tylko jeśli wiązanie typów jest statyczne — lub też dynamiczne. To ostatnie jest teoretycznie możliwe zarówno przy wiązaniu statycznym, jak i dynamicznym. Rzadko jednak zdarza się sprawdzanie dynamiczne przy wiązaniu statycznym — bo czemu odkładać coś co można zrobić przed uruchomieniem programu, jeśli cenimy efektywność jego wykonania?

[14] Fortran (jeśli nie zadeklarujemy typów wprost, co też jest możliwe) przyjmuje, że wszystkie zmienne, których nazwy rozpoczynają się literami `I`–`N` są typu całkowitego, a pozostałe — zmiennoprzecinkowego.

[15] W każdym użyciu zmiennej jej nazwa poprzedzona jest znakiem mówiącym o typie wyrażenia: `$` — zmienna skalarna (prosta, jak liczba); `@` — tablica; `%` — tablica asocjacyjna (patrz Podrozdział 3.3.2.7).

[16] Tam nazwy zmiennych napisowych kończyły się znakiem `$`.

Silne/słabe. Sprawdzanie zgodności typów może być bardzo restrykcyjne. Mówimy wtedy o silnym typowaniu. Oznacza to, że wszelkie (na tyle, na ile to możliwe) niezgodności typów są wyłapywane i raportowane jako błędy.

Wiele języków oferuje jednak słabe typowanie, które polega na tym, że w sytuacji niezgodności typów język decyduje za programistę jak typy dopasować i próbuje to zrobić przez *niejawne konwersje typów*.

Oczywiście sytuacja tutaj nie jest czarno-biała, wiele jest stanów pośrednich pomiędzy słabym i silnym typowaniem. Tak na przykład, PHP jest językiem o bardzo słabym typowaniu, poprawne są w nim na przykład wyrażenia `12+'4 krowy'` oraz `12 . '4 krowy'` (pierwsze z nich ma wartość `16`, drugie — `'124 krowy'`). W podobnej sytuacji jest C (choć tutaj konwersje często oparte są na reprezentacji maszynowej danej — szczególnie widać to w funkcjach z nieustalonymi typami parametrów formalnych — zamiast na jej znaczeniu, co pogarsza sytuację niedbałego programisty). Z drugiej strony mamy Adę, która nie pozwala nawet na niejawną konwersję między typami liczbowymi — wyrażenie `3+4.0` jest niepoprawne! Gdzieś w środku — choć bliżej silnego typowania — znalazłby się język Pascal, ale występują w nim rekordy z wariantami (unie, patrz strona 3.3.2.2), które bardzo osłabiają typowanie (podobnie jak niejawne konwersje).

Sprawę siły typowania komplikują jeszcze wspomniane wyżej niejawne konwersje, które mogą być definiowane w niektórych językach (na przykład jako metody w C++). Tak zdefiniowana niejawna konwersja zawsze obniża — na życzenie programisty — siłę typowania, bo dopuszcza sytuacje, gdzie typy zostaną dopasowane bez wiedzy programisty (a raczej przez jego przeoczenie). W takiej sytuacji nawet wyjściowo silnie typowany język może stać się bardzo słabo typowany, jeśli dopuszcza niejawne konwersje dodawane przez programistę.

Przez nazwę/przez strukturę/przez metody. W końcu dotknąć trzeba semantyki sprawdzania typów, a więc odpowiedzieć na pytanie, kiedy dwa typy uznajemy za zgodne. Najprostszym dla projektantów języka rozwiązaniem (które obowiązuje na przykład w Adzie i w pewnym sensie w Haskellu) jest uznanie za zgodne co do typu tylko tych danych, które zostały zadeklarowane/zdefiniowane/skonstruowane za pomocą tej samej nazwy typu. Jest to jednocześnie rozwiązanie najbardziej restrykcyjne, ale oczywiście najbardziej wrażliwe na wszelkie błędy czy niedbałości programisty, więc często przydatne.

Z drugiej strony mamy sprawdzanie zgodności struktury typów: dwie dane są zgodne jeśli ich typy mają tę samą budowę — tak jest na przykład w języku C. Jest to podejście dość wygodne dla programisty, aczkolwiek czasami zbyt swobodne (nie można odróżnić typów mających tę samą strukturę, ale różne przeznaczenie). No i nie zawsze jednoznaczne. Bo czy do zgodności

struktury dwóch tablic wystarczy równa liczba elementów i ich zgodny typ? A co z zakresem indeksów (jeśli nie ma ustalonego indeksu dolnego, jak to jest w C)? Czy dwie struktury (rekordy) są zgodne, jeśli mają te same pola (co do typów i nazw), ale w różnej kolejności? A co jeśli kolejność typów pól jest ta sama, ale różne nazwy? Na te pytania muszą odpowiedzieć osoby odpowiedzialne za specyfikację języka; przy implementacji takiego języka trzeba także wziąć pod uwagę znacznie większe skomplikowanie porównywania struktury (w stosunku do porównywania nazw).

Zwykle stosowane jest podejście mieszane (jak w Pascalu).

Porównanie tych dwóch podejść pokazują Listingi 3.7 oraz 3.8. W pierwszym z nich, w Adzie wszystkie cztery typy są niezgodne, dzięki temu pisząc na przykład system przeliczający jednostki długości, nie pomylimy nigdy zmiennych zadeklarowanych do przechowywania metrów z tymi do przechowywania stóp. Inaczej jest w C, gdzie typ `metry` jest pod każdym względem zgodny i równoważny typowi `stopy`, podobnie jak typ `TA` typowi `TB`.

Listing 3.7. Zgodność typów w Adzie

```
type metry is new Float;
type stopy is new Float;
type TA is array (1..100) of Integer;
type TB is array (1..100) of Integer;
```

Listing 3.8. Zgodność typów w C

```
typedef float metry;
typedef float stopy;
typedef int[100] TA;
typedef int[100] TB;
```

Całkiem innym podejściem (stosowanym w Pythonie, Smalltalku, Ruby) jest *kacze typowanie* (czyli *typowanie przez sygnatury metod*), o którym więcej w Podrozdziale 7.5.6 na stronie 168.

Związane ze zgodnością nazwy typów są pojęcia *typu pochodnego* i *podtypu*. Na Listingu 3.7 wszystkie zdefiniowane tam typy są typami pochodnymi, czyli typami zdefiniowanymi na podstawie już istniejących, ale niezgodnymi z nimi.

Natomiast podtyp, to typ zgodny z typem, od którego pochodzi (i z innymi jego podtypami!) i może być traktowany jako zawężenie (czyli podzbiór) nadtypu. Specjalnym przypadkiem podtypów są typy okrojone w Adzie czy Pascalu, a także klasy pochodne w wielu językach obiektowych.

3.3.2. Wybrane typy złożone

Typy proste (liczby, znaki, wartości logiczne, wyliczenia...) nie są specjalnie ciekawe i są dość proste zarówno w implementacji, jak i w użyciu. Warto jednak zwrócić uwagę na kilka problemów związanych z implementacją, definiowaniem oraz użyciem *typów złożonych*.

3.3.2.1. Rekordy (struktury)

Typ rekordowy (w niektórych językach: *struktura*) może być rozumiany jako iloczyn kartezjański typów poszczególnych pól, przy czym poszczególne składowe (pola) są identyfikowane własną nazwą (inaczej niż w tradycyjnym iloczynie kartezjańskim, gdzie składowe raczej identyfikuje się kolejnymi liczbami naturalnymi). Typy poszczególny pól mogą być zwykle dowolne i są od siebie niezależne, mogą być więc całkiem różne.

Zaletą rekordów jest możliwość statycznego wyznaczenia adresów pól względem początku danej, bo ramka rekordu[17] jest wyznaczona przez jego definicję.

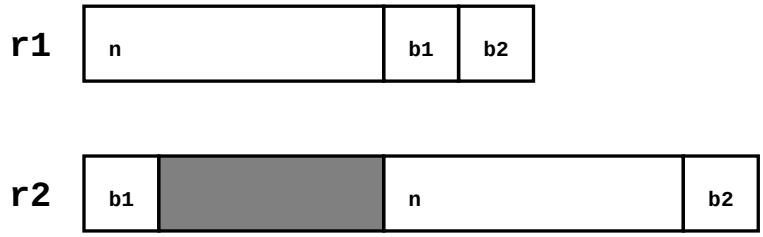
W pamięci rekord zajmuje zwykle zwartą przestrzeń, ale może obejmować też „dziury" — puste bajty wynikające z wymogów architektury sprzętowej, która może nakazywać umieszczanie wartości pewnych typów (na przykład długich liczb całkowitych) pod adresami będącymi wielokrotnością pewnej liczby (2, 4...)[18]. Listing 3.9 przedstawia dwa typy rekordowe różniące się jedynie kolejnością pól w definicji, a Rysunek 3.4 pokazuje jak może wyglądać w pamięci rozmieszczenie zmiennych obu typów przy założeniu, że długie liczby całkowite muszą mieć adres podzielny przez 4.

Listing 3.9. Przykładowe definicje rekordów w Pascalu

```
type Rek1 = record
              n  : LongInt;
              b1 : Byte;
              b2 : Byte;
            end;
     Rek2 = record
              b1 : Byte;
              n  : LongInt;
              b2 : Byte;
            end;
var r1 : Rek1;
    r2 : Rek2;
```

[17] Czyli schemat rozmieszczenia jego zawartości w pamięci — patrz też Rysunek 3.3.

[18] W Pascalu (na przykład) istniała dyrektywa `packed` wstawiana w definicji typu przed słowem `record` (także przed `array`), która zmuszała kompilator do zlikwidowania ewentualnych dziur w pamięci rekordu — oczywiście kosztem efektywności wykonywania na nim działań.



Rysunek 3.4. Możliwe rozmieszczenie w pamięci zmiennych `r1` oraz `r2` z Listingu 3.9

3.3.2.2. Unie

Unie[19] są bardzo pokrewne rekordom[20] — podobnie się je definiuje, tak samo odwołuje się do ich pól... Jednakże, matematycznie typ unijny nie odpowiada iloczynowi kartezjańskiemu typów pól, lecz raczej sumie tych typów. W unii bowiem mamy możliwość korzystania jednoczesnego z tylko jednego z pól — pozostałe są wtedy niedostępne[21].

W praktyce implementowane jest to tak, że wszystkie pola unii zajmują to samo miejsce w pamięci[22]. Niestety, w tradycyjnych uniach nie ma żadnego sposobu sprawdzenia, którego z pól aktualnie używamy. Programista w pełni odpowiada więc za to, by zaglądać do tego pola co trzeba, i nie ma w tej kwestii żadnego wsparcia ze strony języka. Oczywiście, można przewidzieć dodatkową daną, która wskazuje, jakie pole jest aktualnie w użyciu, ale nadal pozostaje to na głowie programisty (patrz Listing 3.10). W związku z tym unie są stosowane obecnie bardzo niechętnie[23] i wiele nowoczesnych języków (Java, C#) w ogóle ich nie dopuszcza[24]. Trzeba wtedy stosować (bezpieczniejsze) zwykłe rekordy.

Listing 3.10. Unia w rekordzie

```
...
typedef union {
  float x;
  int n;
  char z;
```

[19] Analogicznym — choć nie do końca tożsamym — typem jest rekord z wariantami w Pascalu czy też w Adzie.

[20] W języku C na przykład jedyna składniowa różnica między nimi to użycie słowa `union` zamiast `struct`.

[21] A właściwie to, niestety, są dostępne, lecz zaglądanie do nich ujawnia śmieci.

[22] Dokładniej: zaczynają się w tym samym miejscu, bo mogą być różnej długości.

[23] Jedyną racją bytu unii była oszczędność pamięci, ale w dobie coraz tańszych i coraz większych pamięci traci to na znaczeniu.

[24] Szczególnie, że osłabiają typowanie, bo pola unii zajmując to samo miejsce w pamięci pozwalają na niekontrolowany „przeciek” danych z jednego typu do drugiego.

```
6 } UNIA;
  typedef struct {
8   int ktore_pole;
    UNIA u;
10 } REKORD;
  REKORD r;
12 ...
  switch (r.ktore_pole) {
14   case 0:
       zrob_cos_z_rzeczywista(r.u.x);
16     break;
     case 1:
18     zrob_cos_z_calkowita(r.u.n);
       break;
20   case 2:
       zrob_cos_ze_znakiem(r.u.z);
22     break;
     default:
24     zglos_blad();
  }
26 ...
```

Całkiem inaczej ma się sprawa w nowoczesnych językach z silnym typowaniem — patrz Podrozdział 5.6.2.3 na stronie 102.

3.3.2.3. Tablice

Typ tablicowy może być matematycznie utożsamiany z potęgą kartezjańską (czyli z wielokrotnym iloczynem kartezjańskim tego samego zbioru przez siebie) typu elementu. Dostęp do danych przechowywanych w tablicy jest za pomocą indeksów, które mogą być wyrażeniami, co daje możliwość przeglądania w pętli lub też indeksowania pośredniego — bardzo przydatne własności! Jednak z tego samego powodu dostęp jest nieco wolniejszy niż w rekordach, bo musi być wyliczany dynamicznie.

W dzisiejszych językach programowania występuje kilka odmian tablic — ze względu na podejście do zmienności ich rozmiarów.

Rozmiar statyczny. Tablice z rozmiarem ustalonym przed uruchomieniem programu mogą być traktowane jak normalne dane o odpowiednio dużym rozmiarze. Mogą więc być alokowane statycznie, dynamicznie na stosie, czy też dynamicznie na stercie — w zależności od potrzeb.

Rozmiar dynamiczny ograniczony statycznie. Tablice, w których rozmiar maksymalny jest znany przed uruchomieniem programu (a rozmiar aktualny może się zmieniać) mogą być wewnętrznie traktowane jak tablice z rozmiarem statycznym, aczkolwiek w pewnych sytuacjach (na przykład, gdy ograniczenie jest bardzo wysokie) optymalne może być potraktowanie ich jak tablic z rozmiarem w pełni dynamicznym.

Rozmiar dynamiczny stały nieograniczony. Tablice, w których rozmiar jest ustalany dopiero w czasie działania program, ale już się potem nie zmienia, mogą być traktowane jak tablice z rozmiarem statycznym poza jednym wyjątkiem: nie mogą być alokowane statycznie, więc pozostaje tylko stos lub sterta.

Rozmiar dynamiczny zmienny. Najbardziej skomplikowaną sytuację przedstawiają tablice, w których rozmiar może zmieniać się dowolnie. Tutaj muszą one być alokowane na stercie, co więcej należy przewidzieć (kosztowną!) możliwość realokacji pamięci, gdy program będzie wymagał powiększenia bieżącego rozmiaru.

Inną kwestią jest kontrola zakresu dla indeksu (która jest kosztowna, bo musi być przeprowadzana dynamicznie). W językach nowoczesnych jest ona standardem, ale w starszych językach może być wyłączana (jak w Pascalu) lub w ogóle nie istnieje (jak w C, gdzie jest nie tyle problemem co pewną zaletą języka — w połączeniu z traktowaniem tablic identycznie jak wskaźników, patrz Podrozdział 3.3.2.5).

Kolejnym problemem są tablice wielowymiarowe. Pascal i C traktują je jak tablice tablic. Ada rozróżnia tablice tablic od tablic wielowymiarowych (Listing 3.11). Podejście 'tablica tablic' ma w tych językach tę zaletę, że można odwoływać się do poszczególnych tablic składowych (wierszy czy kolumn) jak do osobnych danych (tak jest z tablicą `T2` z tego kodu: `T2(3)(4)` oznacza daną całkowitą, natomiast `T2(7)` — tablicę 10-elementową).

Listing 3.11. Tablice wielowymiarowe w Adzie

```
-- tablica dwuwymiarowa
type T1 is array (1..10, 1..10) of Integer;

-- jednowymiarowa tablica tablic jednowymiarowych
type T2 is array (1..10) of array (1..10) Integer;
```

Jeszcze inne możliwości to operowanie na wycinkach tablic — oferuje je na przykład Fortran czy Python (przy czym w Pythonie tablicom odpowiadają listy — Listing 7.13 na stronie 144) — a także wbudowane działania typu porównywanie, dodawanie czy mnożenie tablic przez liczbę (jak to jest możliwe w różnych pakietach obliczeniowych, czy też w Fortranie).

3.3.2.4. Napisy

Napis w wielu językach (szczególnie starszych) jest tablicą znaków (język C). Niektóre języki traktują ten typ specjalnie wprowadzając dodatkowe możliwe operacje na napisach, jak choćby konkatenacja (Pascal), czy też pozwalając na dynamiczną zmianę rozmiarów napisu — nawet, jeśli nie pozwalają na takie operacje na tablicach. Uzasadnienie jest proste: napisy są

bardzo specjalnymi tablicami — zawsze jednowymiarowe, typ elementu jest jednobajtowy — a ponadto takie operacje są kluczowe dla wielu zastosowań komputerów (choćby przetwarzanie tekstów). W innych językach (szczególnie wysokopoziomowych) implementacja tablicowa napisu jest ukryta przed programistą i typ napisowy traktowany jest jak typ prosty.

Jeśli chodzi o podejście do ustalenia i zmienności rozmiarów napisów, to języki podchodzą do nich różnie, jak w przypadku tablic (Podrozdział 3.3.2.3). Zwykle jednak — zgodnie z tym co napisaliśmy powyżej — napisy są traktowane nieco bardziej „dynamicznie".

3.3.2.5. Wskaźniki

Typy wskaźnikowe są zbiorami wartości oznaczających położenie innych danych w pamięci. W najprostszym wypadku wskaźnik reprezentowany jest przez adres w pamięci, pod którym jest wskazywana dana. Wskaźnik może przechowywać również inne informacje — jak typ danej wskazywanej, adres wirtualny, indeks w tablicy obiektów itp. Te wszystkie dane są jednak zwykle ukryte przed programistą i takie bogatsze typy nazywa się często *typami referencyjnymi*.

Arytmetyka wskaźników. Bez wskaźników nie można obejść się we współczesnym programowaniu — przydatne są do dynamicznego zarządzania pamięcią i korzystania ze sterty. Jest jednak jeden aspekt wskaźników, który zdecydowanie dzieli języki — jest to *arytmetyka wskaźników* używana do elastycznego (i często szybkiego) adresowania pośredniego. Arytmetyka wskaźników polega na tym, że możemy podawać i wyliczać położenie pewnych danych względem innych danych w pamięci — czyli **możemy wskaźniki przesuwać**. Jest obecna w C i C++, a także (choć nieco ukryta) w Pascalu. Z drugiej strony nowsze języki wysokiego poziomu (Java, C#, Python, nie wspominając nawet o językach deklaratywnych, jak Haskell czy Prolog) nie pozwalają na wykonywanie takich operacji i referencje służą tam tylko jako wskaźniki do obiektów — mogą być alokowane, dealokowane, podstawiane im nowe referencje, wyłuskiwane obiekty wskazywane, ale **wskaźniki nie mogą być przesuwane po pamięci**.

Listing 3.12 pokazuje typowe operacje na wskaźnikach w języku C, skupiając się, na arytmetyce wskaźników.

Listing 3.12. Arytmetyka wskaźników w języku C

```
1 #include <stdio.h>

3 int main(void) {
      int t[10];
5     int *p;
      int i;
```

```
7
        p = t;
9
        for(i=0; i<10; i++)
11          p[i] = i*i;

13      printf("%d\n", t[3]);
        printf("%d\n", p[3]);
15      printf("%d\n", *(t+3));
        printf("%d\n", *(p+3));
17      printf("%d\n", 3[t]);
        printf("%d\n", 3[p]);
19
        for(p+=9; p>=t; p--)
21          printf("%d\n", *p);

23      printf("%d\n", t[30]);
        printf("%d\n", (31)[p]);
25      printf("%d\n", (-3)[t]);
        printf("%d\n", *(p-2));
27      printf("%d\n", *(t-13));
        printf("%d\n", p[-12]);
29
        return 0;
31  }
```

Warto zwrócić tu uwagę na kilka kwestii występujących w języku C (i C++), które są tam bardzo ważne, ale często pomijane lub zaniedbywane.

— Wiersz 4 deklaruje tablicę 10 liczb całkowitych.
— Wiersz 5 deklaruje wskaźnik do danej całkowitej.
— Co natomiast dzieje się w wierszu 8? Wygląda to jak podstawienie tablicy pod wskaźnik! Tak jednak nie jest. Trzeba bowiem wiedzieć, że C (prawie) nie rozróżnia tablic i wskaźników. Tak więc wyrażenie `t` jest adresem początku tablicy i jako adres może być do `p` przypisane, bo wskaźniki też są w C po prostu adresami. Jednak na odwrót nie jest to możliwe (Czytelnik na pewno to sprawdzi), bo tablice traktowane są jako wskaźniki stałe — i to jest właściwie jedyna różnica.
— Wiersze 10–11 wypełniają tablicę kwadratami liczb, ale korzystamy tu ze zmiennej `p`, bo podobnie jak tablice mogą być traktowane wskaźnikowo, tak wskaźniki — tablicowo.
— Wiersze 13–18 wyświetlają tę samą wartość (`9`), ale dostęp do niej uzyskują na różne sposoby — wiersze nieparzyste korzystają z tablicy `t`, zaś parzyste ze wskaźnika `p`. Mało tego, o ile wiersze 13–14 wykorzystują dobrze znaną notację tablicową, to wiersze 15–16 korzystają właśnie z arytmetyki wskaźników. Co oznacza wyrażenie `*(t+3)`? Jak wiadomo `*w` oznacza *dereferencję* czyli wyłuskanie danej spod adresu wskazywa-

nego przez `w`. W takim razie, co wskazuje `t+3`? Jest to adres względny przesunięty względem `t` o 3, ale nie bajty, lecz `inty` — bo taki jest typ wskazywany przez `t`.

— Chyba najbardziej jednak zaskakujący zapis pojawia się w wierszach 17–18! Otóż okazuje się, że skoro zapis `a[b]` jest równoważny zapisowi `*(a+b)`, a dodawanie jest przemienne, to także zapisowi `*(b+a)`. W takim razie można spróbować zapisać to wyrażenie jako `b[a]`... Okazuje się, że działa. Zapis `a[b]` jest bowiem tylko *lukrem składniowym*[25] dla bardziej odstraszającego (i o dwa znaki dłuższego) zapisu `*(a+b)`.

— Wiersze 20–21 pokazują, że wskaźniki (tutaj `p`) można przesuwać. W linii 20 ustawiamy wskaźnik na ostatnim elemencie tablicy i w kolejnych obrotach pętli przesuwamy się wstecz, dopóki nie osiągniemy początku tablicy — wskaźniki można więc w C także porównywać: `p>=t`.

— W końcu wiersze 23–28 pokazują to, czego zwykle robić się nie powinno: zaglądanie do pamięci, co do której nie mamy pewności, czy należy do nas[26]. Jest ona poza naszą tablicą (przed lub za). Poszczególne wiersze powinny parami (23–24, 25–26, 27–28) wyświetlać tę samą wartość — lub też przerywać działanie programu z błędem *naruszenia ochrony pamięci* (ang. *segmentation fault*), który jest plagą programistów nieostrożnie posługujących się wskaźnikami (głównie w C i C++). Taka jest też geneza błędu *przepełnienia bufora*[27], co jest często powodem dziur w oprogramowaniu mogących skończyć się naruszeniem bezpieczeństwa systemu.

  Różnice o jeden w kolejnych wierszach wynikają z tego, że po skończeniu pętli z linii 20–21 wskaźnik `p` jest ustawiony nie na początku tablicy, lecz o jeden `int` przed nią.

Kłopoty ze wskaźnikami. Wielu programistom wskaźniki — szczególnie te proste, jak w C/C++/Pascalu — przysparzają wielu zmartwień. Sprawa jest jeszcze gorsza, gdy w użyciu jest arytmetyka wskaźników. Błędy tego rodzaju słynne są z tego, że bardzo trudno je znaleźć, a co gorsza, nie

[25] Lukier składniowy, to taki element składni języka, który można odrzucić bez straty funkcjonalności, ale jest wygodny dla programisty. Pokrewne terminy to *sól składniowa* (cecha składni języka, która zmusza programistę do pisania lepszego kodu, lub utrudnia pomyłki — jak w Adzie słowo kluczowe `end`, które musi zawsze występować ze wskazaniem, co kończy — `end if`, `end while` itp.) oraz *sacharyna składniowa*, *słodzik składniowy* (cecha składniowa języka, której można się pozbyć bez utraty funkcjonalności, ale — w przeciwieństwie do lukru — niczego nie ułatwia).

[26] Jeszcze gorzej byłoby coś w takiej pamięci próbować zapisać!

[27] Przepełnienie bufora (ang. *buffer overflow*) polega właśnie na tym, że programista przewiduje zbyt łagodne lub błędne kontrolowanie indeksu tablicy, gdzie użytkownik może coś zapisywać i pośrednio pozwala w ten sposób wprowadzać do pamięci procesu, ale poza przewidziane do tego miejsce, dane niebezpieczne.

zawsze wychodzą na jaw w zwykłym testowaniu (tylko dopiero w jakichś niesprzyjających okolicznościach).

Jeden z problemów zaprezentowaliśmy w Listingu 3.12 — jest to zaglądanie (lub gorzej: zapisywanie) pod zły adres. Może to prowadzić do naruszenia ochrony pamięci (czyli próby zajrzenia/zapisania do pamięci innego procesu) lub mazania po swojej pamięci — jeśli trafi się na adres jakichś własnych istniejących danych; w najlepszym przypadku będzie to odczytanie bezsensownych danych (jak właśnie w Listingu 3.12).

Kolejny błąd wynikający z przeoczenia programistów (a zdarzający się w najlepszym profesjonalnym oprogramowaniu) to *wyciek pamięci*. Zmienne dynamiczne alokowane jawnie na stercie w wielu językach trzeba także jawnie zwalniać (patrz strona 41 Podrozdział 3.2.2.1), a jeśli dopuszczona jest arytmetyka wskaźników, to innej możliwości praktycznie nie ma. Często jednak programiści zapominają umieścić w odpowiednim miejscu wywołanie funkcji `free`... Prowadzi to oczywiście do tego, że dana pamięć jest już do końca pracy programu zarezerwowana i wielokrotnie powtarzane takie zapomnienie może spowodować wyczerpanie pamięci dostępnej dla procesu i jego awaryjne przedwczesne zakończenie.

Wyciek pamięci jest też skutkiem *zagubienia danych*. Wyobraźmy sobie następujący fragment programu (Listing 3.13).

Listing 3.13. Zagubione dane (język C)

```
1 ...
  int *p;
3 int *q;
  ...
5 p = calloc(20, sizeof(int));
  ... /* tu robimy cos z p */
7 p = calloc(15, sizeof(int));
  ...
```

Jeśli pomiędzy wierszem 5 a 7 nie zwolnimy pamięci zaalokowanej w wierszu 5, ani nie zapamiętamy wskaźnika do tej pamięci w jakiejś innej zmiennej wskaźnikowej, to cała ta 20-elementowa tablica pozostanie w pamięci zaalokowana już na zawsze. Nie będziemy mieć już bowiem do niej żadnego dostępu, bo w wierszu 7 pod `p` podstawiamy nowy wskaźnik wymazując (ale nie dealokując!) stary. Mamy więc wyciek pamięci.

Ale jest gorzej, bo być może zaszła pomyłka i w wierszu 7 miała być zmienna `q`, nie `p`, bo tablica z wiersza 5 miała być nadal w użyciu. Wtedy mamy do czynienia z zagubionymi danymi, bo owa tablica jest w pamięci, ale bezpowrotnie stracona...

No i z tego kodu może wyniknąć jeszcze przepełnienie bufora, bo jeśli coś

będziemy chcieli zapisać do komórki `p[18]` wierząc, że jest to ciągle tablica 20-elementowa, wpiszemy to w pamięć niewiadomego przeznaczenia.

To nie koniec kłopotów ze wskaźnikami. Może zdarzyć się sytuacja w pewnym sensie odwrotna do powyżej opisanych (Listing 3.14).

Listing 3.14. Wiszący wskaźnik (język C)

```
    ...
2   int *p;
    int *q;
4   ...
    p = calloc(100, sizeof(int));
6   q = p;
    ... /* tu robimy cos z p */
8   free(p);
    p = NULL; /* niekonieczne, ale dla bezpieczenstwa */
10  ...
```

Jeżeli nic nie robimy ze zmienną `q` pomiędzy liniami 6 a 8, to po linii 9 zmienna `q` nadal wskazuje na obszar pamięci, na który wskazywała wcześniej — jednak teraz jest on już zwolniony, co w dużym programie może być łatwe do przeoczenia. Tak więc wszelkie operacje na danych wskazywanych przez `q` są błędne — `q` jest teraz *wiszącym wskaźnikiem* (ang. *dangling reference*).

Jeśli dodatkowo po linii 9 wykonamy `free(q)`, to mamy *podwójne zwolnienie* (ang. *double free*), czyli drugi raz zwalniamy pamięć już zwolnioną, co jest błędem[28].

Zbieranie nieużytków. Na szczęście wszystkie te problemy mogą zostać względnie prosto rozwiązane — o ile tylko zrezygnujemy z arytmetyki wskaźników[29]

Zbieranie nieużytków (także *odśmiecanie*, a po angielsku *garbage collection*) polega na takim zarządzaniu alokacją pamięci na stercie, by można było wykrywać i zwalniać obszary zaalokowane, ale już nieużywane. Wszystkie języki wysokiego poziomu (na przykład Java, C#, Python, Haskell, Prolog, Lisp, Scheme), które oferują niejawną alokację danych na stercie, muszą jakiś sposób ich niejawne dealokacji mieć w postaci zbierania nieużytków. Jest kilka odmian zbierania nieużytków, dwie (podstawowe) z nich tutaj pokrótce omówimy.

[28] Dlatego w linii 9 podstawiamy `NULL` pod `p` — żeby nie zrobić podwójnego zwolnienia na `p`. Oczywiście, dla `q` trzeba by zrobić osobne takie podstawienie, jeśli się o tym pamięta...

[29] Po przeczytaniu najbliższych akapitów Czytelnik na pewno sam odkryje, dlaczego zbieranie nieużytków i arytmetyka wskaźników stoją w opozycji. Nie będziemy psuć zabawy podpowiedziami.

Najprostszym sposobem — a jednocześnie całkiem szybkim i działającym na bieżąco — jest *zliczanie referencji*. Podejście to przypomina nieco sytuację znaną z systemów plików (takich jak Linuksowe *ext2* czy *ext3*). Otóż każdy obiekt alokowany na stercie ma własny wewnętrzny licznik referencji prowadzących do tego obiektu (od innych). W momencie utworzenia obiektu licznik ten wynosi jeden. Za każdym razem, gdy powstaje nowe wskazanie z innego obiektu do rozważanego licznik referencji zwiększa się o jeden. Za każdym razem, gdy takie wskazanie jest zrywane — zmniejsza się o jeden. Kiedy licznik osiągnie zero, oznacza to, że obiekt już potrzebny nie jest i jego pamięć może być zwolniona. Dzieje się to na bieżąco w czasie alokacji i zrywania wiązań. Listing 3.15 oraz Rysunek 3.5 pokazują użycie licznika referencji w praktyce w hipotetycznym języku (`napis` oraz `rekord` to konstruktory alokujące pamięć i inicjujące nowe obiekty, `NIC` to wskaźnik pusty, służący do zerwania referencji; prostokątami zaokrąglonymi oznaczyliśmy nazwy zmiennych, prostokątami ostrymi — obiekty, trójkąt to licznik referencji, strzałki przedstawiają referencje, krzyżyki — zerwane referencje).

Listing 3.15. Zliczanie referencji w praktyce

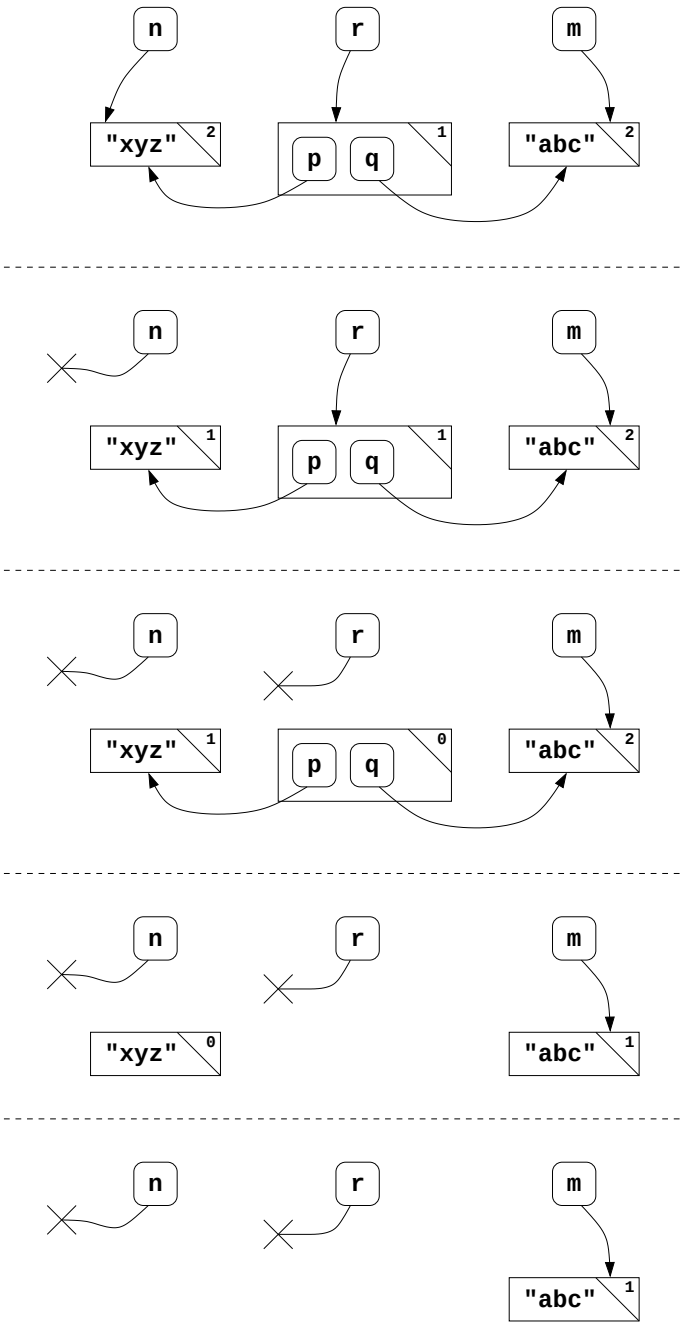
```
  n = napis("xyz")
2 r = rekord(p, q)
  r.p = n
4 r.q = napis("abc")
  m = r.q
6 n = NIC
  r = NIC
```

Niestety, Listing 3.16 oraz Rysunek 3.6 pokazują, że są sytuacje (związane z cyklicznymi strukturami danych), kiedy może dojść do wycieku pamięci nawet przy zliczaniu referencji. Tu niestety liczniki referencji zawodzą i programista musi wiedzieć, by przed ostatecznym zerwaniem wiązania do struktury cyklicznej przerywać także ją ręcznie.

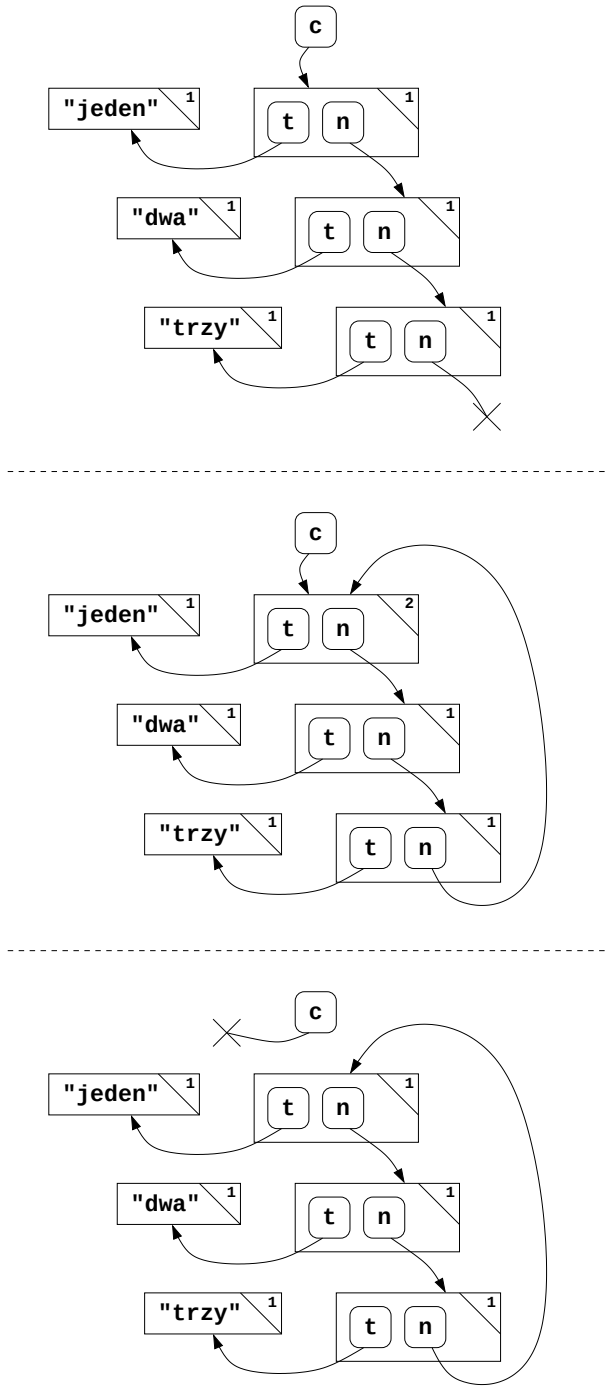
Listing 3.16. Zliczanie referencji dla listy cyklicznej

```
1 c = rekord(t, n)
  c.t = napis("jeden")
3 c.n = rekord(t, n)
  c.n.t = napis("dwa")
5 c.n.n = rekord(t, n)
  c.n.n.t = napis("trzy")
7 c.n.n.n = c
  c = NIC
```

Inny sposób to *oznaczanie-i-zamiatanie* (ang. *mark-and-sweep*). Także wymaga, by każdy obiekt miał jakiś znacznik (choć tu wystarczy jeden bit).



Rysunek 3.5. Sytuacja na stercie w Listingu 3.15; patrząc od góry: po linii 5; po linii 6; po linii 7 (etap I); po linii 7 (etap II); po linii 7 (etap III)



Rysunek 3.6. Sytuacja na stercie w Listingu 3.16, od góry: po linii 6; po linii 7; po linii 8

Proces ten odbywa się (w przeciwieństwie do zliczania referencji) raz na jakiś czas, zwykle wtedy, gdy pamięć jest na wyczerpaniu. Składa się on (w zarysie) z trzech kroków:

1. przejrzenie całej sterty i oznaczenie wszystkich obiektów jako nieużywanych;
2. trawersowanie rekurencyjne wszystkich struktur po wskaźnikach, poczynając od zewnątrz (na przykład od tablicy nazw spoza sterty) i oznaczanie wszystkich napotkanych obiektów jako używanych[30];
3. przejrzenie całej sterty i dealokacja wszystkich obiektów oznaczonych nadal jako nieużywane.

Sposób ten jest w pełni skuteczny, ale wymaga większego nakładu pracy oraz postępowania rekurencyjnego (punkt 2 powyżej). W związku z tym wymaga też pamięci na stosie, a skoro jest on wywoływany wtedy, gdy pamięci zaczyna brakować, może to być problematyczne. Jeszcze jedną kwestią jest widoczne zawieszanie się pracy programu na czas odśmiecania w bliżej nieprzewidywalnym momencie, bo zwykle jednak zajmuje ono trochę czasu — nie nadaje się więc to w zastosowaniach, gdzie czas rzeczywisty jest ważny. Niemniej jednak, sposób ten jest najpopularniejszy współcześnie.

3.3.2.6. Listy

Kilka słów jeszcze należy powiedzieć o *listach*, które od czasów Lispa [34] zyskały dużą popularność jako struktury wbudowane (a wręcz podstawowe) w językach deklaratywnych (Lisp, Scheme, Haskell, Prolog)[31].

Lista w Lispie (a w pozostałych wymienionych wyżej językach jest analogicznie) jest zdefiniowana rekurencyjnie, jako:

— `nil`, czyli wskazanie puste;
lub
— `(cons` *`głowa ogon`*`)`, gdzie `cons` jest konstruktorem pary złożonej z dowolnych dwóch danych; przy czym *`głowa`* może być dowolną daną, natomiast *`ogon`* powinien być listą (inaczej całość nie będzie listą).

Z listy można standardowymi selektorami wyłuskać głowę (`car`) oraz ogon (`cdr`)[32].

[30] Tutaj — aby nie wpaść w nieskończoną rekurencję — po napotkaniu obiektu oznaczonego jako używany nie idziemy w głąb, bo wiadomo, że było to miejsce już odwiedzone.

[31] W Pythonie listy mają nieco inne własności — patrz Podrozdział 7.3.1.3 na stronie 143.

[32] Albo mówiąc inaczej: `car` wyłuskuje pierwszy element pary, a `cdr` — drugi. Przy okazji warto wiedzieć, że nazwy tych funkcji są właściwie przypadkowe i pochodzą z maszyny, na której implementowano w 1959 roku pierwsze wersje Lispa (IBM 704) — tam cała para mieściła się w 36-bitowym słowie (rejestrze) podzielonym na dwie części: adresu i dekrementacji; stąd skróty od: *contents of address of register* oraz *contents of decrement of register*.

A więc prawdziwe są równości[33] pokazane na Listingu 3.17.

Listing 3.17. Listy i operacje na nich w Lispie

```
'()              == nil
'(a)             == (cons a nil)
'(a b)           == (cons a '(b))
                 == (cons a (cons b nil))
'(a b c d)       == (cons a (cons b (cons c (cons d nil))))
'(a (b c) d)     == (cons a (cons '(b c) (cons d nil)))
  == (cons a (cons (cons b (cons c nil)) (cons d nil)))
(car '(a))                          == 'a
(cdr '(a))                          == nil
(car '(a b c d))                    == 'a
(cdr '(a (b c) d))                  == '((b c) d)
(car (car (cdr '(a (b c) d))))      == 'b
```

Ważną cechą takiej listy jest jej asymetria: wyłuskanie głowy następuje w czasie stałym, niezależnym od długości listy; natomiast wyłuskanie ostatniego elementu — w czasie wprost proporcjonalnym do długości listy. Dlatego też wszelkie algorytmy listowe pisane są „od głowy” (więcej w Rozdziałach 5 oraz 6).

3.3.2.7. Tablice asocjacyjne

Na koniec rozdziału zostawiliśmy wygodne i elastyczne struktury danych, jakimi są *tablice asocjacyjne* zwane też *słownikami* oraz *tablicami haszującymi*. Są one z natury dynamiczne i matematycznie stanowią zbiór par uporządkowanych postaci (k, w), gdzie k jest kluczem identyfikującym parę (a więc nie może być dwóch par o tym samym kluczu w jednym słowniku) i należy do wybranego przez projektantów języka zestawu typów[34], natomiast w jest dowolną wartością. Od strony programisty wykorzystującego takie tablice, działają one jak zwykłe tablice dynamiczne, ale indeksowane są kluczami, niekoniecznie indeksami liczbowymi.

Trzy poniższe Listingi (3.18–3.20) pokazują użycie tablic asocjacyjnych w trzech językach programowania.

Listing 3.18. Tablica asocjacyjna w Perlu (za [37])

```
%wzrost = ("Jacek"  => 177,
```

[33] Na Listingu 3.17 równości są zapisane infiksowo, w prawdziwym Lispie wszystko zapisywane jest prefiksowo. Używamy za to poza tym Lispowej notacji z nawiasami i apostrofem — lista bez apostrofu oznacza aplikację funkcji (pierwszego elementu) do pozostałych elementów jako argumentów; natomiast apostrof przed listą oznacza jej nieobliczanie (traktowanie literalne).

[34] Na przykład w PHP są to napisy i liczby, a w Pythonie wszelkie dane głęboko niezmienialne. Przy okazji: w PHP nie ma zwykłych tablic — wszystkie są asocjacyjne.

```
2             "Joanna" => 166,
              "Jerzy"  => 199);
4 $wzrost{"Jola"} = 188;
  delete $wzrost{"Jerzy"};
6 %wzrost = ();
  if (exists $wzrost{"Joanna"}) ...
```

Listing 3.19. Tablica asocjacyjna w PHP

```
1 $wzrost = array("Jacek"  => 177,
                  "Joanna" => 166,
3                 "Jerzy"  => 199);
  $wzrost["Jola"] = 188;
5 unset($wzrost["Jerzy"]);
  $wzrost = array();
7 if (isset($wzrost["Joanna"])) ...
```

Listing 3.20. Tablica asocjacyjna w Pythonie

```
1 wzrost =   {"Jacek"  : 177,
              "Joanna" : 166,
3             "Jerzy"  : 199}
  wzrost["Jola"] = 188
5 del $wzrost["Jerzy"]
  wzrost = {}
7 if "Joanna" in wzrost: ...
```

3.4. Pytania i zadania

1. Wyjaśnij pojęcia: wiązanie, czas wiązania, wiązanie statyczne, wiązanie dynamiczne, ramka wywołania podprogramu, okres życia danej, okres życia wiązania, zakres widoczności, poprzednik statyczny, poprzednik dynamiczny, przesłonięcie, aliasowanie.
2. Jakie są różnice między dynamicznym i statycznym zakresem widoczności?
3. Jakie są zalety i wady wiązań statycznych, a jakie — dynamicznych?
4. Jakie dane mogą być alokowane statycznie, jakie na stosie, a jakie na stercie?
5. Na jakie etapy można podzielić powstawanie języka i oprogramowania ze względu na tworzenie się wiązań?
6. Z Listingu 3.21 wypunktuj możliwie dużo różnych wiązań (minimum 20 powinieneś znaleźć, ale raczej jest dużo więcej) i podziel je na statyczne i dynamiczne, biorąc pod uwagę, że język C jest językiem kompilowanym.

Listing 3.21. Wiązania w C

```c
#include <stdio.h>
#define N 10
int main(void) {
  int i;
  for(i=1; i<=N; i++)
    printf("%d   %d   %d", i, i*i, i*i*i);
  return 0;
}
```

7. Jakie są zalety i wady wiązań dynamicznych, a jakie statycznych?
8. Co to jest dana? Wymień i scharakteryzuj krótko jej atrybuty.
9. Jak dzielimy dane ze względu na okres ich życia i miejsce alokacji? Jakie jest zastosowanie, wady i zalety poszczególnych rodzajów?
10. Kiedy mówimy o językach wysokiego poziomu, a kiedy o językach niskiego poziomu?
11. Wyjaśnij, dlaczego w Fortranie nie można było do niedawna używać rekurencji.
12. Uszereguj języki według siły typowania: Pascal, Ada, C, Haskell, PHP, Python.
13. Jakie może być typowanie (sprawdzanie zgodności typów)?
14. Skąd mogą się brać dziury w reprezentacji rekordów w pamięci operacyjnej?
15. Dlaczego tradycyjne unie uchodzą za niebezpieczne?
16. Wyjaśnij, dlaczego nie da się pogodzić arytmetyki wskaźników i zbierania nieużytków.
17. Wypunktuj i omów podejścia do implementacji rozmiaru tablic.
18. Jakie są podobieństwa, a jakie różnice między tablicami a listami?
19. Ile elementów ma Lispowa lista `'(a (b (c d) e f) (()) ())`?
20. Wymień i omów problemy związane ze wskaźnikami alokowanymi i dealokowanymi jawnie przez programistę.
21. Znajdź przykładowe zastosowania dla tablic asocjacyjnych.
22. Jak rozumieć w języku C wyrażenie `3[t]`, gdzie `t` jest tablicą?
23. Jak może być implementowana dealokacja niejawna?
24. Jakie są wady i zalety zliczania referencji w porównaniu do oznaczania-i-zamiatania?

Rozdział 4

Podprogramy i programowanie obiektowe

4.1. Podprogramy

Podprogramy są znane w językach programowania także pod różnymi innymi nazwami: procedury, funkcje, metody, operacje, operatory, działania, współprocedury, współprogramy, generatory, predykaty... — w zależności od języka i konkretnych własności podprogramu.

Bez względu na nazwę, łączy je to, że podprogramy są abstrakcją działań podejmowanych przez komputer — podobnie jak dane (a węziej — zmienne) są abstrakcją pamięci. Jednakże w odróżnieniu od zmiennych podprogramy są znacznie[1] starsze, bo nawet bardzo prymitywne (choć nie wszystkie) kody maszynowe znają ideę podprogramów[2], a zmiennych tam może nie być, tylko bezpośredni dostęp do pamięci.

4.1.1. Pojęcia podstawowe

Zaczniemy od krótkiego omówienia kilku podstawowych pojęć związanych z podprogramami.

Punkt wejścia podprogramu. Jest to miejsce w kodzie, od którego podprogram się rozpoczyna. Języki programowania powszechnie nie dopuszczają wielu punktów wejścia do jednego podprogramu[3]. Wynika to zwykle stąd, że potrzebne są na początku pewne czynności organizacyjne (patrz niżej, strona 68, Podrozdział 4.1.2), więc skok do wnętrza podprogramu (i tym samym przeskoczenie owych czynności) może doprowadzić do błędnego działania podprogramu lub nawet całego programu. W asemblerach jednak taki skok do wnętrza podprogramu jest możliwy, a jego dopuszczalność wynika z tego, że programista i tak powinien wszystkiego dopilnować ręcznie.

Oczywiście, można wyobrazić sobie język programowania, w którym można zadeklarować w podprogramie wiele miejsc, od których się zaczyna, a wywołanie określałoby takie miejsce. Kompilator (czy interpreter) mógłby wtedy w tych wszystkich miejscach przewidzieć wykonanie owych czynności organizacyjnych. Zaburza to jednak bardzo czytelność programu i jest niepraktyczne w językach wysokiego poziomu, bo nie daje tu żadnej oszczędności (która może być jedyną motywacją takiego postępowania w kodzie maszynowym).

[1] Jak na taką młodą dziedzinę jak informatyka...

[2] Jeśli jest stos, tą są tam zwykle rozkazy `CALL` oraz `RET`. Nawet jeśli ich nie ma, a jest stos, to podprogramy można ręcznie, przez zwykłe skoki i odkładanie adresów na stosie, zaimplementować (choć to niewygodne).

[3] Aczkolwiek współprocedury oraz generatory dopuszczają wejście w miejscu ich ostatniego opuszczenia — więcej w Podrozdziale 7.6.3 na stronie 176.

Punkt wyjścia podprogramu. Jak jest punkt wejścia, to jest i punkt wyjścia. Jest to miejsce w kodzie, w którym podprogram kończy swe działanie. W przeciwieństwie do punktów wejścia, zwykle[4] dopuszcza się wiele punktów wyjścia, które załatwia instrukcja zwykle zwana `return`.

Wywołanie podprogramu. Jest to po prostu uruchomienie (wykonanie) podprogramu przez inną jednostkę (inny podprogram lub program główny).

Definicja podprogramu. Program — żeby mógł być wywołany (czyli uruchomiony) — musi zostać zdefiniowany[5]. Definicja podprogramu składa się zwykle z dwóch zasadniczych części — nagłówka i ciała podprogramu.

Ciało podprogramu. Jest to właśnie kod, który opisuje działanie tego podprogramu, i który jest wykonywany w czasie jego wywołania.

Nagłówek podprogramu. Jest on znakiem składniowym dla parsera, że rozpoczyna się tu definicja podprogramu. Ponadto nagłówek podaje *nazwę podprogramu*[6], która identyfikuje podprogram w jednostce i pozwala się do niego odwoływać, przede wszystkim w wywołaniu. W końcu zadaniem nagłówka jest ustalenie liczby, kolejności, nazw i typów parametrów formalnych podprogramu — oraz ewentualnego typu wyniku podprogramu.

Parametry formalne podprogramu. Parametry formalne (zwane też po prostu *parametrami*) to nazwy pojawiające się w deklaracji podprogramu. Służą jego parametryzacji i w ciele podprogramu mogą być używane jak zwykłe zmienne lokalne (choć czasami z pewnymi ograniczeniami — zależnie od modelu i implementacji przekazywania danych, patrz niżej). Kilka ciekawych rodzajów składni deklarowania parametrów formalnych (zmienna liczba parametrów, parametry domyślne) ilustruje język Python (Podrozdział 7.4.1 na stronie 153), choć w innych językach można podobne rozwiązania także spotkać (na przykład C, C++, Ada, PHP).

Parametry aktualne podprogramu. Z drugiej strony, wyrażenia przekazywane podprogramowi w jego wywołaniu nazywane są parametrami aktualnymi (także: *argumentami*). Są to dane, z którymi na czas bieżącego wywołania wiązane są parametry formalne i konkretyzują je w danym wykonaniu ciała podprogramu. Sposobów (składniowych) tego powiązania jest

[4] Ale nie w oryginalnym Pascalu, gdzie jedynym punktem wyjścia był koniec podprogramu.

[5] Nie muszą być definiowane podprogramy, które nigdy nie będą uruchomione, takie jak metody abstrakcyjne. Wystarczyć może wtedy tylko deklaracja.

[6] Ale są też funkcje anonimowe — patrz Podrozdziały 5.3, 5.7, 7.6

kilka, niektóre z nich omówimy w praktyce na przykładzie języka Python (Podrozdział 7.4.2 na stronie 157).

Sygnatura podprogramu. Sygnatura podprogramu jest to liczba i kolejność (a w niektórych językach także nazwy oraz typy) parametrów formalnych podprogramu.

Protokół podprogramu. Protokół podprogramu jest natomiast specyfikacją sposobu kontaktu ze „światem zewnętrznym" (z innymi jednostkami: programami i podprogramami). Składa się on więc z sygnatury (opisującej parametry formalne) oraz z typu wyniku (jeśli podprogram wynik zwraca).

Deklaracja podprogramu. W pewnych sytuacjach (metody abstrakcyjne, podprogramy znajdujące się w osobnym pliku/module/bibliotece, zapowiedzi podprogramów) nie ma konieczności — a czasem wręcz nie wolno — definiować podprogramów, ale trzeba o nich kompilator/interpreter powiadomić. Robi się to przez deklarację podprogramu (nie: definicję), czyli podanie informacji niezbędnych do jego wywołania, a dokładniej: do zastosowania w odpowiednim kontekście. Deklaracja musi podać więc protokół podprogramu (patrz powyżej). Odbywa się to zwykle przez podanie samego nagłówka podprogramu z ewentualnym dopiskiem mówiącym o tym, że ciała nie będzie (`forward` w Pascalu, `;` w C i C++, `=0;` w C++, `abstract` w Javie i C#) lub w określonym kontekście, gdzie ciała nie można dopisać (definicja interfejsu w Javie, specyfikacja pakietu w Adzie, interfejs pakietu w Pascalu). Taką deklarację nazywa się także czasem *prototypem*.

4.1.2. Przebieg wywołania podprogramu

Ciało podprogramu to kod, który ma być wykonany, gdy podprogram jest wywołany. Nie są to jednak jedyne czynności związane z wywołaniem podprogramu — W rzeczywistości mamy trzy fazy wykonywania podprogramu, które składają się z wielu czynności: *inicjalizacja* (w tym *prolog podprogramu*), wykonanie kodu i *finalizacja* (w tym *epilog podprogramu*). Określenia 'prolog' i 'epilog' odnoszą się jedynie do czynności wykonywanych przez podprogram (po skoku do niego, ale przed powrotem), ale nie zapisanych jawnie w kodzie podprogramu. Natomiast 'inicjalizacja' i 'finalizacja' to pojęcia szersze, zawierające czynności wykonywane także przez jednostkę wywołującą (a więc przed skokiem do podprogramu oraz po powrocie).

Warto zwrócić uwagę, że ze względu na stosową implementację podprogramów, czynności epilogu odpowiadają czynnościom prologu, ale w (mniej więcej) odwrotnej kolejności.

Inicjalizacja podprogramu. Przed wykonaniem ciała podprogramu musi dokonać się kilka czynności organizacyjnych. Poniższa lista przedstawia możliwą kolejność (choć nie jedyną — niektóre elementy mogą być opuszczone, dodane, zmieniona kolejność; zależy to od implementacji).

— Na stosie muszą być zapamiętane wartości wszelkich tych rejestrów procesora, które mogą się zmienić w podprogramie, a których wartości mogą być potrzebne później, po powrocie z podprogramu.
— Obliczane są wartości parametrów aktualnych i wkładane w odpowiedniej kolejności na stos (być może także do wyróżnionych rejestrów, jeśli potrzeba szybkiego działania[7]. Rezerwowane jest także miejsce na wyniki (w tym na parametry wyjściowe).
— Inne specjalne dane są umieszczane na stosie jako parametry ukryte — o ile jest taka potrzeba (między innymi obsługa dostępu do zmiennych nielokalnych pochodzących z jednostki wywołującej, ale nie jest to potrzebne dla zmiennych globalnych, które są — jak pamiętamy ze strony 38, Podrozdział 3.2.2.1 — statyczne).
— Wykonywany jest odpowiedni rozkaz maszynowy (w asemblerach zwykle zwany `CALL`), który odkłada na stos aktualny adres wykonywanej instrukcji maszynowej, po czym skacze do adresu, pod którym zaczyna się podprogram.

Teraz zaczyna się prolog podprogramu.

— Podprogram alokuje dla siebie ramkę na stosie zgodnie z potrzebną przestrzenią (patrz Rysunek 3.3 na stronie 40) — na zmienne lokalne i podobne dane.
— Podprogram kładzie na stosie rejestry, które mogą być potrzebne jednostce wywołującej, a mogą zostać nadpisane.

Finalizacja podprogramu. W końcu po wykonaniu ciała podprogramu dokonują się różne czynności kończące. Ta lista znowu jest tylko jedną z wielu możliwości, aczkolwiek inne będą raczej podobne.

— Podprogram wkłada wartość wynikową (wartości wynikowe) w przeznaczone do tego miejsce (zarezerwowane uprzednio przez jednostkę wywołującą).
— Podprogram odtwarza ze stosu rejestry, które tam włożył.
— Podprogram zwalnia ramkę ze stosu tak, by na szczycie znalazł się adres powrotu.
— Podprogram wywołuje instrukcję maszynową powrotu z podprogramu (zwykle `RET`), która pobiera ze stosu adres i skacze podeń, wznawiając tym samym pracę jednostki wywołującej.

Tu kończy się epilog podprogramu.

[7] Zawsze potrzeba.

— Zwalniany jest obszar stosu przeznaczony na parametry, przy czym wartości wynikowe przenoszone są do ich miejsc docelowych.
— Rejestry odtwarzane są ze stosu.

4.1.3. Przekazywanie danych

Powyżej opisaliśmy bardzo skrótowo jak technicznie dane są przekazywane — za pośrednictwem stosu[8]. Nie mówi to jednak nic o tym, co z tymi danymi może robić podprogram. Musimy rozważyć modele i bardziej szczegółowo implementację przekazywania parametrów.

4.1.3.1. Modele semantyczne przekazywania danych

Wszystkie parametry formalne występujące w definicji podprogramu można podzielić na trzy grupy, według semantyki przekazywania danych pomiędzy procedurą a jednostką ją wywołującą.

Tryb wejściowy. Tryb ten (zwany też z angielska trybem *in*) służy do tego, by jednostka wywołująca nasz podprogram mogła przekazać dane **do** podprogramu, ale już nie z powrotem. Tak więc wartość parametru aktualnego musi być w jakiś sposób utożsamiona z parametrem formalnym w momencie wywoływania podprogramu.

Tryb wyjściowy. Ten tryb (zwany też trybem *out*) to przeciwieństwo poprzedniego — służy do przekazania danych **z** podprogramu do jednostki wywołującej. Tutaj więc wartość parametru aktualnego musi być związana z parametrem formalnym w momencie powrotu z podprogramu.

Tryb wejściowo-wyjściowy. W końcu tryb, który łączy oba powyższe — pozwala przekazywać dane w obie strony. (inna nazwa to oczywiście *in-out*).

4.1.3.2. Implementacja przekazywania danych

Powyższe modele dają nam wgląd w wybór możliwości i pozwalają na podjęcie odpowiedniej decyzji przy definiowaniu podprogramu co do wyboru trybu dla danego parametru formalnego. Czasem jednak ważne są szczegóły techniczne implementacji danego trybu w danym języku. Poniżej podajemy możliwe sposoby realizacji różnych trybów przekazywania danych do i z podprogramów.

[8] Oczywiście, wiele języków dopuszcza techniczną możliwość przekazywania danych i wyników także przez zmienne globalne (czyli przez efekty uboczne). Jest to jednak w ogólności niezalecane, bo — choć bardzo efektywne — prowadzi do utraty czytelności programów i łatwo generuje trudne do wykrycia błędy.

Analizując poniższe przypadki warto pamiętać, że tak czy owak w przekazywaniu parametrów bierze udział stos (z pomocą, być może, rejestrów procesora), więc zawsze coś na stos jest wkładane przed skokiem do podprogramu (w przypadku parametrów wejściowych) i/lub ze stosu zdejmowane po powrocie z podprogramu (w przypadku parametrów wyjściowych).

Przekazywanie przez wartość. Ten sposób przekazywania danych polega na włożeniu na stos przed skokiem do podprogramu wartości danych przekazywanych. Oznacza to **skopiowanie** tych danych z ich miejsca pierwotnego do miejsca przeznaczonego na stosie na parametry. W związku z tym ów parametr formalny działa wewnątrz podprogramu **całkowicie jako zwykła zmienna lokalna**, zainicjowana tylko odpowiednią wartością parametru aktualnego. I tak jak każda zmienna lokalna jest bez śladu usuwany po zakończeniu podprogramu.

Widać, że ten sposób realizacji przekazywania wartości nadaje się jedynie do trybu wejściowego. Jest idealny dla niewielkich danych (jak dane proste, niewielkie tablice, napisy, rekordy). Jednak dla dużych danych może być kosztowny. Dana bowiem jest duplikowana — więc jest to zarówno kosztowne z punktu widzenia pamięci zajmowanej przez daną dwukrotnie[9], jak i czasu potrzebnego na skopiowanie danej, zwykle proporcjonalnego do jej rozmiaru.

Przekazywanie przez wynik. Przekazywanie przez wynik jest procesem odwrotnym do poprzedniego: przed wywołaniem podprogramu na stosie musi zostać zarezerwowane miejsce dla danych wyjściowych, a przed powrotem podprogram musi skopiować tam obliczoną wartość wynikową[10], którą po powrocie jednostka wywołująca kopiuje do miejsca docelowego.

Ten sposób — dla odmiany — to realizacja trybu wyjściowego. Znowu, jak poprzednio i z tych samych powodów, jest on bardzo efektywny dla małych danych, natomiast ociężały dla dużych.

Mimo symetrii z poprzednim, z tym trybem wiążą się pewne nowe problemy, których w poprzednim nie ma. Wynikają one z asymetrii pomiędzy parametrami formalnymi a aktualnymi. Otóż nie może być (zwykle) dwóch parametrów formalnych o tej samej nazwie, natomiast mogą być dwa identyczne parametry aktualne, lub też wpływające jeden na drugi. Co bowiem będzie wynikiem działania programu z Listingu 4.1 (język hipotetyczny, który realizuje tryb wyjściowy przez wynik; pomysł za [37])? Zależy to od kolejności kopiowania parametrów. Specyfikacja języka powinna rozwiązywać takie niejednoznaczności w sposób niebudzący wątpliwości.

[9] Dodatkowo pamiętać trzeba, że stos bywa pamięcią „bardziej kosztowną" od sterty.

[10] Alternatywnie i pewnie częściej ze względu na efektywność — obliczenia są wykonywane od razu w owym zarezerwowanym miejscu.

Listing 4.1. Niejednoznaczności wynikające z przekazywania danych przez wynik

```
...
procedure P(out a, out b)
   a := 1;
   b := 2;
end;
...
P(x, x);
print(x);
...
T[1] := 10;
T[2] := 20;
T[3] := 30;
i := 3;
P(i, T[i]);
print(i, T[1], T[2], T[3]);
...
```

Przekazywanie przez wartość i wynik. Jest to połączenie dwóch poprzednich trybów — dane są najpierw kopiowane w jedną stronę, a potem w drugą. Zalety i wady także są te same. Jest to oczywiście realizacja trybu wejściowo-wyjściowego.

Ta realizacja nosi często miano *przekazywania przez kopiowanie*, choć zdarza się także, że wcześniejsze dwie także tak się nazywa (czasem z dodatkowym określeniem, które mówi, czy to dane, czy wyniki są kopiowane).

Przekazywanie przez referencję. Jest to całkiem inny sposób realizacji trybu wejściowo-wyjściowego. Sprowadza się on do przekazania przez wartość (czyli skopiowania na stos do parametru formalnego, a potem używania jako zmiennej lokalnej — patrz wyżej) referencji do danej. Natomiast podprogram działać może dzięki temu na danej bezpośrednio, bo ma do niej wskaźnik.

Dzięki temu jest to dużo szybsze od kopiowania, a i problemy przy trybie wyjściowym są „łagodniejsze" (gdyby tryb wyjściowy był realizowany przez referencję w Listingu 4.1, wyniki byłyby jednoznaczne — choć może niekoniecznie zgodne z zamysłem autora procedury `P`, ze względu na aliasowanie — patrz strona 36, Listing 3.3).

Sposób ten może też być stosowany dzięki temu jako efektywna realizacja trybu wejściowego oraz trybu wyjściowego — o ile kompilator (interpreter) będzie zezwalał jedynie na odczyt parametru formalnego (w przypadku trybu wejściowego) lub na jego zapis (w przypadku trybu wyjściowego).

Leniwe przekazywanie. Ostatni ze sposobów[11] przekazywania parametrów też może realizować tryb wejściowo-wyjściowego (lub jeden z dwóch pozostałych, jeśli język będzie ograniczał w definicji podprogramu odczyt lub zapis do parametru formalnego). Tu jednak samo przekazanie odbywa się całkiem inaczej, bowiem **wartość parametru aktualnego nie jest obliczana**, lecz przekazywana do podprogramu jest jego postać dosłowna[12]. Dalsze postępowanie może w szczegółach się różnić, ale wszystko sprowadza się do tego, że to podprogram w takiej (nieobliczonej) postaci wstawia je w miejsce parametru formalnego i dopiero w razie potrzeby wylicza (patrz też strona 103, Podrozdział 5.7).

4.1.3.3. Przykłady

Poniżej kilka języków wraz z przeglądem zaimplementowanych w nich sposobów przekazywania parametrów.

Ada. Modelowy przykład języka, w którym trzy tryby są oznaczane jawnie wprost (`in`, `out`, `in out`). Dodatkowo kompilator zakazuje odczytu zmiennych trybu `out` i zapisu do zmiennych trybu `in`. Dzięki takiemu podejściu przekazywanie może być optymalizowane, bo duże dane mogą być bezpiecznie przekazywane przez referencję.

Fortran. Tryb wejściowo-wyjściowy realizowany przez referencję. Nowsze wersje pozwalają na wybór trybu (`in`, `out`, `inout`).

C. Tylko tryb wejściowy przez wartość. Można jednak ręcznie przekazać referencję przez wartość i w ten sposób mamy tryb wejściowo-wyjściowy. Można też (kwalifikator `const`) przekazywać wskaźniki jedynie wejściowo (kompilator zabroni podstawiać pod zmienne wskazywane), ale efektywnie. Ponadto — o czym często się zapomina — makra preprocesora C mają leniwe (choć w dość prymitywny sposób) przekazywanie parametrów!

C++. Tu mamy jak w C, ale dodatkowo pojawia się przekazywanie przez referencję.

Java, Python, Ruby, Scheme. Zawsze przez referencję[13].

[11] Jest to właściwie cała rodzina sposobów, z których można wymienić między innymi *przekazywanie przez nazwę*, które to często utożsamiane jest z leniwym przekazywaniem.

[12] Albo, co prawie na jedno wychodzi, postać prekompilowanego przepisu na obliczenie jego wartości.

[13] Niektórzy upierają się tutaj, że jest to przekazywanie przez wartość, bo wszystkie zmienne w tych językach mają wartość referencji do obiektów. Inni stosują tu mało popularne określenie *call-by-sharing* ukute przez Barbarę Liskov [72], oznaczające właśnie przekazywanie wartości do referencji obiektów.

Pascal. W tradycyjnym Pascalu mamy dwa tryby: wejściowy jest realizowany przez wartość, wejściowo-wyjściowy przez referencję.

Haskell. Jak inne języki czysto funkcyjne, Haskell operuje zaawansowanym leniwym przekazywaniem.

4.1.3.4. Podprogramy jako parametry podprogramów

Przy okazji omawiania przekazywania parametrów nie można pominąć problemu przekazywania podprogramów jako parametrów do innych podprogramów. Oczywiście, dają taką możliwość języki funkcyjne, bo jest ona jednym z fundamentów programowania funkcyjnego (Rozdział 5), ale spotykana także jest (często w okrojonej postaci) w innych językach (Pascal, C, C++, Python i wiele innych). Typowym przykładem jest potrzeba napisania funkcji `calkaNum` realizującej całkowanie numeryczne i przyjmującej cztery parametry: początek i koniec przedziału całkowania, dokładność całkowania oraz funkcję zmiennoprzecinkową jednoargumentową, która ma być scałkowana. Możemy wtedy posługiwać się taką funkcją jak na Listingu 4.2 (gdzie `f`, `g`, `sin` są uprzednio zdefiniowanymi przez autora programu lub w bibliotekach funkcjami o jednym parametrze zmiennoprzecinkowym i takimż wyniku).

Listing 4.2. Funkcja jako parametr innej funkcji (w C++)

```
...
cf = calkaNum(-1.0, 0.0, 0.0001, f);
csin = calkaNum(-PI, PI, 0.000001, sin);
...
cout << calkaNum(-2.0, 2.0, 0.001, g);
...
```

Z przekazywaniem podprogramów wiąże się jedno pojęcie — *szzzzzzzzzrzodowisko@środowisko referencyjne*. Jest to ogół wiązań aktywnych w danym momencie programu. Rozważmy program z Listingu 4.3 (język hipotetyczny, `main` to program główny). Jaki będzie wynik tego programu i dlaczego?

Listing 4.3. Środowisko referencyjne

```
int x := 1;

procedure g()
  print(x);
end;

procedure h(f)
  int x := 2;
```

```
      f()
10  end;

12  procedure main()
      int x := 3;
14    h(g);
    end;
```

Mamy tutaj trzy różne zmienne `x`. Wynikiem może być wyświetlenie `1`, `2` lub `3` w zależności od tego, jakie środowisko referencyjne jest używane dla procedury `g` w wierszu 14.

Jeśli mamy do czynienia z *wiązaniem głębokim*, oznacza to użycie wiązań z wiersza 3, czyli z **momentu zdefiniowania** funkcji `g`. Tak więc wyświetlona zostanie liczba `1`. Sytuacja taka jest naturalna dla języków ze statycznym zakresem widoczności (jak C, C++, Pascal).

W przypadku *wiązania płytkiego* wyświetlona zostanie liczba `2`, bo użyte są wiązania z wiersza 9, czyli z **momentu użycia** funkcji `g` (tu oczywiście jako wartość parametru formalnego `f`). Tak będzie się działo w językach o dynamicznym zakresie widoczności (jak Python).

Jest jeszcze trzecia możliwość — gdyby wziąć pod uwagę wiązanie z wiersza 14, to dostalibyśmy wynik `3`. Jest to *wiązanie ad hoc* (liczy się **moment przekazania** funkcji) i nie jest używane w żadnym powszechnym języku.

4.1.4. Polimorfizm podprogramów

Na koniec zagadnień związanych z podprogramami pozostał ich *polimorfizm*, zwany też *przeciążaniem podprogramów*. Polega on na tym, że jedna nazwa służy wielu podprogramom. Możemy więc wyróżnić:

— *polimorfizm dynamiczny*, inaczej *polimorfizm obiektowy*, ściśle związany z programowaniem obiektowym i szerzej omówiony w Podrozdziale 4.2; bardzo podobne jest też *kacze typowanie* (Podrozdział 7.5.6);

— *polimorfizm statyczny ad hoc*, czyli „zwyczajne" podprogramy przeciążone, które mogą być skojarzone z odpowiednim kodem podczas kompilacji i są rozpoznawane na podstawie sygnatury (C++) lub protokołu (Haskell);

— *polimorfizm statyczny*, związany z programowaniem rodzajowym; przykładem takiego programowania są szablony w C++ czy też pakiety rodzajowe w Adzie albo klasy typów w Haskellu; związanie nazwy z odpowiednim kodem także jest tutaj statyczne (w czasie kompilacji), ale nie na podstawie sygnatury/protokołu lecz konkretyzacji (a więc podania brakujących typów) szablonu/pakietu wprost przez programistę przed użyciem; bardzo wygodny, umożliwia wielokrotne wykorzystanie jednego kodu.

4.2. Programowanie obiektowe

Programowanie obiektowe[14] ujrzało światło dzienne w latach 60. za sprawą języka Simula 67 [44] uznawanego, za pierwszy język obiektowy. Kolejnym — już tym razem czysto obiektowym (czyli takim, w którym wszystko jest obiektem) — był Smalltalk, a potem programowanie obiektowe zdobyło nagłą i szeroką popularność dzięki językowi C++, a w ostatnim czasie popularność utrzymuje się i większe bądź mniejsze elementy paradygmatu obiektowego są dostępne w językach takich jak Java, Ada, C#, JavaScript, Object Pascal, Python, Ruby...

Podstawową cechą charakterystyczną programowania obiektowego jest **związanie w jednym bycie danych z operacjami na nich wykonywalnymi**.

4.2.1. Pojęcia

Z programowaniem obiektowym związanych jest wiele pojęć wymagających tutaj wyjaśnienia.

Hermetyzacja. Zwana jest także *enkapsulacją*. Oznacza ona możliwość tworzenia *abstrakcyjnych typów danych*[15], czyli takich, w których mamy wyraźnie oddzielony jawny *interfejs* (czyli zestaw operacji publicznych możliwych do wykonania) od ukrytej *implementacji* (czyli wewnętrznej realizacji owych operacji).

Dziedziczenie. Pozwala ono na budowanie jednych klas na bazie drugich, tak, by w sposób naturalny jeden typ danych był specjalnym przypadkiem innego typu danych.

Polimorfizm dynamiczny (obiektowy). Polimorfizm pozwala na dynamiczne — czyli w czasie pracy programu — dopasowywanie odpowiednich metod do konkretnych danych.

Klasa. Klasa jest abstrakcyjnym typem danych, definiowanym programowo przez podanie operacji dopuszczalnych na nim oraz przez jego reprezentację i implementację tych operacji.

[14] Niektórzy wolą mówić 'programowanie obiektowo zorientowane'. Tym bardziej, że można programować w sposób zorientowany obiektowo bez obiektów.

[15] W programowaniu obiektowym klasy są typami abstrakcyjnymi; typy abstrakcyjne mogą też występować w językach nieobiektowych (przykład: Haskell). Nie należy ich jednak mylić z klasami abstrakcyjnymi, które są dużo węższym pojęciem.

Obiekt. Obiekt jest instancją klasy czyli inaczej mówiąc po prostu elementem danego typu.

Klasa pochodna, klasa potomna, podklasa. Klasa wywiedziona przez dziedziczenie z innej klasy zwana jest klasą względem niej pochodną.

Klasa bazowa, nadklasa. Odwrotnie, jeśli `A` jest klasą pochodną względem `B`, to `B` jest klasą bazową względem `A`.

Metoda. Metoda to każdy podprogram działający na obiektach klasy, który jest zdefiniowany w ramach tej klasy.

Pole. Pole to zmienna zamknięta wewnątrz obiektu. Spotyka się też pojęcia: własność obiektu, zmienna obiektu.

Komunikat. Komunikat to wywołanie metody.

Protokół obiektu, protokół komunikatów obiektu. Jest to zestaw metod obiektu (a właściwie: zestaw protokołów metod), czyli zbiór wszelkich sposobów odwołania się do obiektu.

4.2.2. Sterowanie dostępem

Z hermetyzacją związane jest ściśle sterowanie dostępem do obiektów danej klasy i osiąga się je przez kwalifikatory przy metodach i polach. Zwykle wyróżnia się trzy:

— `public` przy metodzie lub polu powoduje, że jest ono publiczne, czyli dowolnie dostępne z zewnątrz, z każdego innego obiektu;
— `private` działa odwrotnie: ukrywa przed wszystkimi innymi obiektami i klasami daną metodę lub pole — ten właśnie kwalifikator dostępu umożliwia tworzenie realnych typów abstrakcyjnych;
— `protected` jest w pewnym sensie pośredni pomiędzy powyższymi dwoma: ujawnia element dla klas pochodnych, ale przed innymi ukrywa.

Różne języki różnie traktują elementy bez kwalifikatorów — na przykład w Pythonie w ogóle brak takich kwalifikatorów i wszystko jest traktowane zawsze publicznie. W wielu innych językach domyślna jest kwalifikacja publiczna dla metod, a prywatna dla pól.

4.2.3. Dziedziczenie

Dziedziczenie polega na zdefiniowaniu nowej klasy (a więc nowego typu) w oparciu o pewną inną klasę. W klasie pochodnej można dodać nowe pola i nowe metody, można też *zmodyfikować* metodę zmieniając jej protokół

(o ile język na to pozwala) lub też *przedefiniować*, czyli zmienić jedynie jej ciało, nie zmieniając protokołu.

Dziedziczenie może być *pojedyncze* (po jednej klasie bazowej), ale niektóre języki (C++, Python) zezwalają na dziedziczenie *wielokrotne*. Niestety, taka możliwość bywa uważana za patologię, bowiem powoduje pewne niejednoznaczności. Rysunek 4.1 pokazuje przykład dziedziczenia wielokrotnego. Powstają wtedy pytania, co oznacza zapis `x.pole` lub `x.metoda`, gdy obiekt `x` jest jest obiektem klasy `X`? Ile (licząc te odziedziczone) w klasie `X` jest składowych o nazwach `pole`, `metoda`, `jedno` i do których odnoszą się podane zapisy? Szczególnie, że tutaj klasy `A`, `B`, `C`, `D` tworzą *przerażający diament* (ang. *diamond of dread*), więc nawet zapis `x.jedno` nie jest całkiem jasny! Ponadto — co może być problemem poważniejszym — miesza się w ten sposób implementacje wielu klas, a może to prowadzić do bałaganu.

Dlatego też niektóre języki nie dopuszczają dziedziczenia wielokrotnego (na przykład Java). Wtedy jednak traci się funkcjonalność związaną z tym, że pewna klasa może implementować wiele typów abstrakcyjnych. Dlatego też w Javie (i innych językach) zamiast dziedziczenia wielokrotnego mamy możliwość implementowania *interfejsów*. Interfejs to rodzaj klasy abstrakcyjnej, która (jak to klasa abstrakcyjna, patrz niżej) nie może mieć instancji, ale opisuje pewien zestaw metod, który inne klasy mogą zaimplementować. Stanowi więc rodzaj *kontraktu*, który obiecują wypełnić inne klasy.

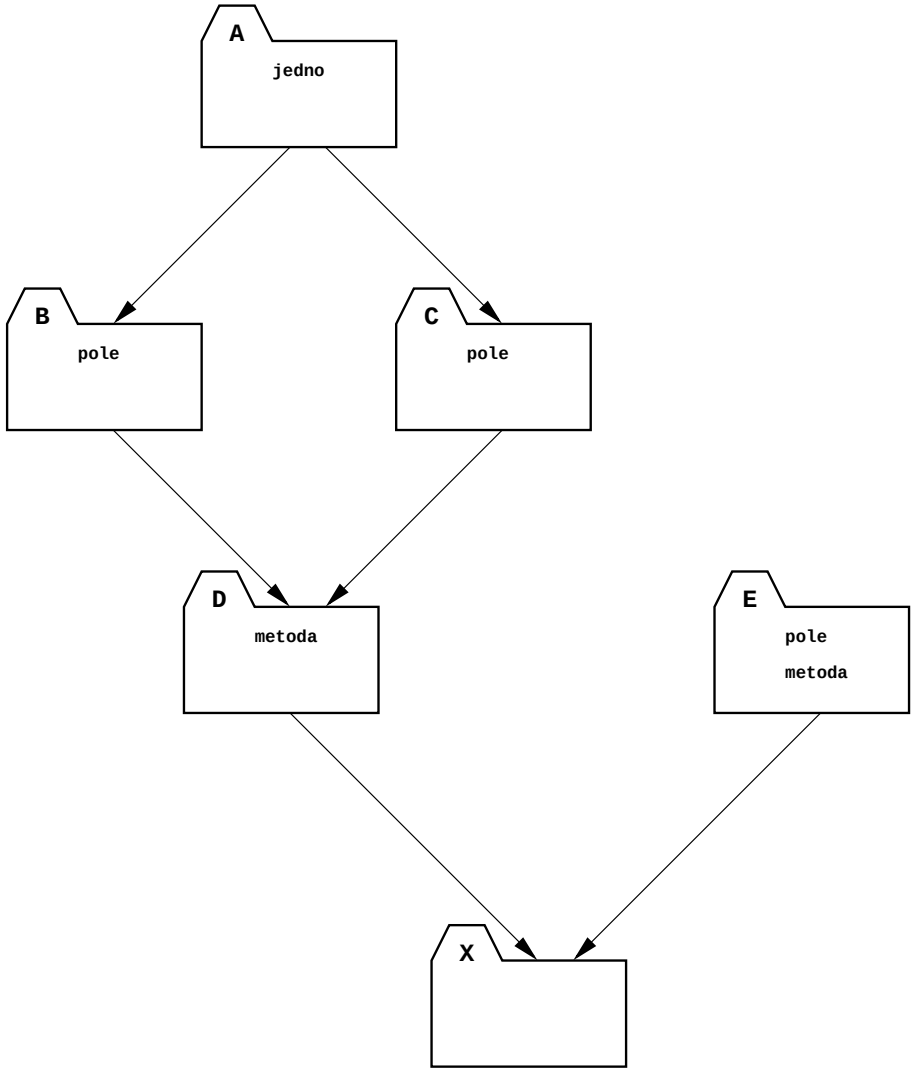
Z drugiej strony, języki z dynamicznym systemem typów (jak Python) mimo, że dopuszczają dziedziczenie wielokrotne, nie cierpią z powodu mieszania interfejsu z implementacją — właśnie dzięki dynamicznemu typowaniu.

Wspomniana wyżej *klasa!abstrakcyjna* to taka klasa, która nie ma instancji, ale deklaruje (nie definiuje!) pewne metody (*metody abstrakcyjne*), które mogą być zaimplementowane w klasach pochodnych. Służy jedynie (podobnie jak interfejs[16]) za rodzaj szablonu dla swoich podklas. W przeciwieństwie do interfejsu nie wszystkie metody klasy abstrakcyjnej muszą być abstrakcyjne — wystarczy jedna.

4.2.4. Podklasa jako podtyp, polimorfizm

Ważnym zagadnieniem jest określenie przez język, czy podklasy są podtypami (w sensie ze strony 47, Podrozdział 3.3.1). Jeśli tak, to możemy zmiennym (także parametrom formalnym podprogramów) zadeklarowanym jako nadklasowe nadawać wartości z ich podklas. W takiej sytuacji nie może być możliwości zmodyfikowania metody, jedynie jej przedefiniowanie (czy-

[16] Jednak — w odróżnieniu od interfejsu — klasa abstrakcyjna może mieć także metody nieabstrakcyjne.



Rysunek 4.1. Przykładowy graf dziedziczenia klas — z dziedziczeniem wielokrotnym (duże litery to klasy, małe to ich pola lub metody, strzałki prowadzą od nadklasy do podklasy)

li zachowanie protokołu). Podklasy jako podtypy umożliwiają polimorfizm dynamiczny obiektowy.

Żeby go uzyskać klasy bazowe definiują *metody wirtualne*, które nie tylko mogą być przedefiniowane przez podklasy, ale zmienne zadeklarowane jako nadklasowe będą wywoływać te metody wirtualne, które odpowiadają realnemu typowi danej przechowywanej w zmiennej, nie zaś typowi bazowemu.

Żeby obsłużyć metody wirtualne, każda dana obiektowa klasy mającej takie metody musi oprócz zwykłych pól (przechowywanych w *rekordzie instancyjnym*) zawierać także referencję do *tablicy metod wirtualnych* (*vtable*), przygotowywanej dla każdej klasy w czasie kompilacji programu. Tablica ta zawiera referencje do konkretnych metod wirtualnych danego typu. Takie postępowanie jest konieczne tylko w językach ze statycznym typowaniem — przy dynamicznym typowaniu wszystko i tak jest ustalane w czasie pracy programu, taka tablica jest więc zbędna.

4.3. Pytania i zadania

1. Wyjaśnij pojęcia: podprogram, punkt wejścia, punkt wyjścia, wywołanie, definicja podprogramu, deklaracja podprogramu, ciało podprogramu, nagłówek podprogramu, parametry formalne, parametry aktualne, sygnatura podprogramu, protokół podprogramu, inicjalizacja, prolog, finalizacja, epilog, polimorfizm, obiekt, klasa, hermetyzacja, dziedziczenie, podklasa, nadklasa, vtable, metoda, komunikat, protokół obiektu, sterowanie dostępem, interfejs, klasa abstrakcyjna, metoda abstrakcyjna, metoda wirtualna, dziedziczenie wielokrotne, przerażający diament.
2. Jakie czynności implementacyjne wiążą się z rozpoczęciem, a jakie z zakończeniem wykonywania podprogramu?
3. Na czym polega przeciążanie podprogramów? Jakie są jego rodzaje?
4. Co to jest `CALL` oraz `RET`?
5. Jakie są modele semantyczne przekazywania parametrów? Jak można implementować przekazywanie parametrów?
6. W jakich językach parametrem podprogramu może być podprogram? Jaki paradygmat programowania bazuje na tej możliwości?
7. Co to jest środowisko referencyjne? Jakie może być używane w jakich językach przy przekazaniu podprogramu do podprogramu?
8. Dlaczego dziedziczenie wielokrotne jest często krytykowane? Do czego może się przydać? Jak uzyskać podobny efekt, gdy język nie dopuszcza dziedziczenia wielokrotnego?
9. Czym różni się modyfikacja od przedefiniowania metody?
10. Czym różni się klasa abstrakcyjna od interfejsu? Czy każdy typ abstrakcyjny jest klasą abstrakcyjną?

Rozdział 5

Programowanie funkcyjne

5.1. Trochę historii

Program w języku funkcyjnym można traktować jako zestaw definicji funkcji w sensie matematycznym (i ewentualnie innych danych[1]), z ewentualną wyróżnioną funkcją, którą trzeba obliczyć — jako głównym zadaniem programu[2].

Języki funkcyjne mają swoją już dość długą historię, bo najstarszy z nich — Lisp [34] (współtworzony przez Johna McCarthy'ego [76]), ciągle w użyciu! — został zaprojektowany i zaimplementowany już w latach 50. XX wieku. Jednak historia języków funkcyjnych jest dłuższa, bo są one oparte na opracowanym przez Alonza Churcha [71] λ-rachunku [6, 7], który ujrzał światło dzienne w latach 30. XX wieku i sam może być uważany za pierwszy język funkcyjny (implementowany zresztą jako podjęzyk w każdym z używanych języków funkcyjnych — i nie tylko).

Początkowo wiele języków funkcyjnych nie zachowywało czystości funkcyjnej, zawierając w sobie także elementy imperatywne (przede wszystkim: dostęp do stanu maszyny oraz interakcje z użytkownikiem), albo odwrotnie — nie zawierały żadnych narzędzi związanych ze stanem maszyny i jej zewnętrzem, albo korzystanie z tego rodzaju narzędzi było skrajnie niewygodne. Sytuacja zmieniła się po zastosowaniu monad [62] (Podrozdział 5.8).

5.2. Cechy charakterystyczne języków czysto funkcyjnych

Pozytywne cechy specyficzne programowania funkcyjnego[3] krótko opisujemy poniżej.

Przezroczystość referencyjna. Inaczej: *przezroczystość odwołań*. Polega ona na tym, że wyrażenie mające tę własność może być w każdym swoim użyciu zastąpione swoją wartością. Typowym przykładem wyrażeń przezroczystych referencyjnie są aplikacje funkcji w matematyce: jeśli jakaś funkcja $f : A \mapsto B$ jest matematycznie dobrze zdefiniowana i zachodzi $f(a) = b$, to wszędzie gdzie jest $f(a)$ możemy wstawić b bez zmiany sensu wyrażeń; tak więc następujące wyrażenia są równoważne: $g(b, b)$, $g(f(a), b)$, $g(b, f(a))$, $g(f(a), f(a))$.

W językach nieprzezroczystych referencyjnie jest inaczej — na przykład w Pythonie (patrz też Rozdział 7), który nie jest językiem czysto funkcyjnym, wyrażenie

[1] Bo funkcje w programowaniu funkcyjnym są danymi!

[2] Podobnie jak w innych językach programowania — na przykład w C — możemy opuścić ową wyróżnioną funkcję (w Haskellu, podobnie jak w C byłaby to funkcja o nazwie `main`) i wtedy taki plik traktowany jest (być może po dodatkowej obróbce lub dodaniu pewnych elementów składniowych) jako biblioteka definicji.

[3] Ściślej: czysto funkcyjnego, które nas w tym rozdziale interesuje.

`int(raw_input("Podaj liczbę całkowitą: "))`
nie jest przezroczyste referencyjnie. Tak więc jeśli jego wartość podstawimy pod zmienną `x`, a potem wykonamy instrukcję `print (x+x)`, komputer zada użytkownikowi jedno pytanie i wypisze podwojoną wartość odpowiedzi. Gdybyśmy — wiedząc, co poda użytkownik — zastąpili `x` konkretną wartością (na przykład 12), to dostajemy instrukcję `print (12+12)`, która wypisze `24` nie pytając o nic użytkownika. Z drugiej strony, jeśli zastąpimy `x` odpowiednim wywołaniem funkcji, dostaniemy

```
print (int(raw_input("Podaj liczbę całkowitą: "))
+int(raw_input("Podaj liczbę całkowitą: ")))
```

co oczywiście spowoduje dwukrotne zapytanie użytkownika o liczbę (być może różną za każdym razem) i wypisanie sumy odpowiedzi.

Trwałe struktury danych. Trwałe struktury danych to takie, które nie są zmieniane w trakcie przetwarzania i mogą być wielokrotnie wykorzystywane. Mogą też być fragmentami współdzielone, co znacznie polepsza efektywność ich przetwarzania (zarówno czasową jak i przestrzenną). W końcu dzięki trwałości struktur można nie odróżniać samej struktury od referencji do niej, co przyczynia się do jaśniejszego zapisu.

Wszystkie operacje więc w językach czysto funkcyjnych są *konstruktorami* (ewentualnie *selektorami*), nie zaś *transformatorami*.

Automatyczne zarządzanie pamięcią. Jak wiele języków wyższego poziomu, języki funkcyjne same zarządzają pamięcią przez jakiś rodzaj zbierania nieużytków (patrz strona 56, Podrozdział 3.3.2.5) — tym bardziej, że trwałe struktury danych i tak tego wymagają, a programista i tak nie ma bezpośredniego dostępu do stanu maszyny (w tym do pamięci).

Leniwa ewaluacja. W językach czysto funkcyjnych można (i tak jest zwykle domyślnie) obliczać wartości wyrażeń leniwie[4], czyli w ostatniej chwili, gdy są już potrzebne (patrz też strona 72, Podrozdział 4.1.3.2).

Kolejność czynności. Z leniwą ewaluacją i przezroczystością referencyjną związana jest też kolejność wykonywania czynności obliczeniowych. W programowaniu imperatywnym jest ona z góry określona przez układ programu, natomiast w programowaniu czysto funkcyjnym interpreter/kompilator sam może wybierać czynności do wykonania tak, by było to optymalne. Ułatwia to bardzo programowanie niesekwencyjne (Rozdział 8).

Struktury potencjalnie nieskończone. Leniwe obliczenia umożliwiają definiowanie struktur, które są, matematycznie rzecz biorąc, nieskończone (choć oczywiście dostęp do ich elementów w praktyce jest realizowany

[4] W językach imperatywnych dominuje *gorliwa ewaluacja*, czyli obliczanie wyrażeń jak tylko zostaną napotkane.

w miarę potrzeb, więc nigdy nie trzeba całej takiej struktury w pamięci fizycznie przechowywać).

Silne typowanie. Większość języków czysto funkcyjnych jest bardzo silnie typowana, co tutaj znaczy, że każda dana (także nieznana przed uruchomieniem programu) może zostać w trakcie kompilacji (także częściowej) zakwalifikowana do konkretnego typu lub klasy typów. Można więc przeciążać statycznie funkcje i także statycznie sprawdzać wszelkie niezgodności typów.

Funkcje jako wartości pierwszego rzędu. Funkcje są zwykłymi danymi, więc inne funkcje mogą na nich działać, jak i zwracać je jako wyniki.

Zwięzłość. Zapis funkcyjny programu jest zwykle bardziej zwięzły niż imperatywny (aczkolwiek, jeśli się z tym przesadzi, to może stać się nieczytelny).

Matematyczna ścisłość. Dzięki czystości funkcyjnej programy funkcyjne są ścisłe i dobrze zdefiniowane w sensie matematycznym, co daje możliwość prostego (czasem niemal automatycznego) dowodzenia poprawności programów, a także utrzymania tej poprawności przy jego modyfikowaniu/rozszerzaniu.

Malkontenci doszukają się zapewne także wad w programowaniu czysto funkcyjnym, ale zwykle przesadzają. Można jednak wymieniać tu poniższe cechy.

Trudność w wyrażeniu niektórych operacji imperatywnych. Niektóre operacje — jak interakcja z zewnętrzem (plik, użytkownik, urządzenia we/wy) czy zmiany/uwzględnianie pewnych globalnych danych/stanów/ustawień (liczby pseudolosowe, nietrwałe struktury danych, wyjątki, efekty uboczne) — nie mogą same w sobie być zaimplementowane w postaci funkcji przezroczystych referencyjnie (patrz przykład na stronie 82, Podrozdział 5.2).

Istnieją jednak metody ich prezentowania przez pewne złożenia funkcji działających na określonych typach (monady) lub łańcuchy implikacji (jak w programowaniu logicznym).

Nieefektywność dużych struktur z dostępem swobodnym. Ponieważ nie mamy do dyspozycji transformatorów, zmiany w dużej macierzy reprezentowanej strukturą trwałą muszą tworzyć nową macierz, co jest bardzo nieefektywne, bo polega na skopiowaniu większości poprzedniej (macierze zwykle w drobnym stopniu mogą być współdzielone).

Da się to jednak ominąć wspomnianymi w punkcie poprzednim sposobami.

Brak iteracji. Nie ma w językach funkcyjnych takich czynności[5], jak pętle. W związku z tym wszelkie powtórki muszą być definiowane rekurencyjnie — podobnie jak w programowaniu logicznym (Rozdział 6). Wiadomo przy tym, że rekurencja potrafi być dużo bardziej nieefektywna od iteracji.

Na szczęście — także dzięki przezroczystości referencyjnej — możemy korzystać z *rekurencji ogonowej* (strona 91, Podrozdział 5.5), która może być obliczona efektywnie (to jest bez użycia stosu i kolejnych wywołań podprogramu).

5.3. Rachunek lambda

Rachunek lambda (także: λ-rachunek) jest formalnym systemem definiowania i aplikowania (czyli obliczania) funkcji. Występuje w dwóch pokrewnych wersjach: z typami (prostszy, silniejszy obliczeniowo, ale sprzeczny w sensie teorii logicznej[6]) i bez typów (słabszy obliczeniowo, bardziej skomplikowany, logicznie niesprzeczny). Można go traktować jako prosty język programowania i ma swoje implementacje w językach funkcyjnych (ale nie tylko — także w językach: Python, patrz Podrozdział 7.6, Eiffel, C#, Smalltalk i innych). Zwykle (ze względu na cechy wymienione wyżej) stosowany jest λ-rachunek bez typów, którym się tu zajmiemy.

Wyrażenia λ-rachunku (λ-*wyrażenia*) mogą być trojakiego rodzaju:

— jeśli x jest zmienną, to jest też λ-wyrażeniem;

— jeśli A jest λ-wyrażeniem oraz x jest zmienną, to $\lambda x.A$ jest λ-wyrażeniem nazywanym λ-*abstrakcją*[7];

— jeśli A i B są λ-wyrażeniami, to AB jest λ-wyrażeniem zwanym *aplikacją*[8].

Można dodatkowo używać nawiasów, dla ujednoznacznienia zapisu[9].

Zmienne występujące w każdym λ-wyrażeniu możemy podzielić na związane (te, które występują między znakami λ i . (kropka); odpowiada to parametrom formalnym funkcji) oraz wolne (pozostałe). Kilka zmiennych

[5] Bo funkcje matematyczne właściwie nie opisują czynności, lecz zależności!

[6] Nie jest natomiast sprzeczny jako sposób wyrażania obliczeń, a więc jest dobrym językiem programowania.

[7] λ-abstrakcja może być interpretowana jako definicja anonimowej funkcji.

[8] Aplikacja może być interpretowana jako obliczenie/wywołanie funkcji A dla argumentu B.

[9] Domyślnie, bez użycia nawiasów, abstrakcję uznaje się za silniejszą od aplikacji, a aplikacja jest lewostronnie łączna. Czyli: $abc\lambda d.def = (((ab)c)(\lambda d.((de)f)))$

może mieć tę samą nazwę, bo zmienne mogą się przesłaniać — zmienna związana w danej abstrakcji ma zasięg tylko w niej[10].

Przykład 5.1. Rozważmy λ-wyrażenie $x(\lambda x.x)(\lambda x.(\lambda x.x)x)x$. Występuje tu jedna nazwa x, ale na oznaczenie czterech różnych zmiennych. Przemianowując je (α-konwersja, patrz niżej) możemy uzyskać λ-wyrażenie α-równoważne: $x(\lambda x_1.x_1)(\lambda x_2.(\lambda x_3.x_3)x_2)x$.

Znaczenie każdego λ-wyrażenia można określić przez zdefiniowanie zestawu redukcji (inaczej: konwersji). Opiszemy tu dwie (α i β) z trzech (bo pozostaje jeszcze η).

Najprostsza z nich, *α-konwersja* oznacza przemianowanie zmiennych związanych (ale tak, by nie powstały między zmiennymi kolizje). Jeśli jakieś λ-wyrażenie a można zamienić za pomocą α-konwersji na inne b, to mówimy, że a i b są α-równoważne. Co więcej, zwykle uważa się je po prostu za równe[11]. Patrz Przykłady 5.1 oraz 5.2

Przykład 5.2. Następujące λ-wyrażenia są sobie α-równoważne: $\lambda x.(x\lambda x.x)$, $\lambda x.(x\lambda y.y)$, $\lambda y.(y\lambda x.x)$, $\lambda a.(a\lambda b.b)$. Natomiast $\lambda y.(y\lambda x.y)$ nie jest im α-równoważne.

Druga z konwersji, *β-redukcja*, mówi o tym, jak należy rozumieć obliczenie funkcji. Aplikacja λ-wyrażenia $(\lambda x.A)$ do λ-wyrażenia B (czyli $(\lambda x.A)B$) po przekształceniu przez β-konwersję daje wyrażenie A z zamienionymi wszystkimi wystąpieniami zmiennej związanej x na wyrażenie B.

Przykład 5.3. Zachodzą poniższe β-redukcje (ostatnie dwie, przy założeniu, że `2`, `7`, `*`, `+` są jakimiś λ-wyrażeniami[12]):

$$(\lambda x.x)y \stackrel{\beta}{\Rightarrow} y, \tag{5.1}$$

$$(\lambda x.z)y \stackrel{\beta}{\Rightarrow} z, \tag{5.2}$$

$$(\lambda x.(\lambda y.x\texttt{*}y))\texttt{2} \stackrel{\beta}{\Rightarrow} (\lambda y.\texttt{2*}y), \tag{5.3}$$

$$((\lambda x.(\lambda y.x\texttt{*}y))\texttt{2})(\lambda z.\texttt{7+}z) \stackrel{\beta}{\Rightarrow} \texttt{2*}(\lambda z.\texttt{7+}z). \tag{5.4}$$

[10] Nie chcemy tego definiować tutaj formalnie, bowiem dla informatyka (nawet początkującego) te reguły powinny być intuicyjnie jasne — przykłady powinny w tym pomóc

[11] Intuicja matematyczna czy programistyczna stojąca za tym jest jasna: jeśli zdefiniujemy funkcję z parametrami formalnymi a, b, c, to równie dobrze możemy to zrobić z parametrami *pies*, *kot*, *mysz* i funkcje będą we wszystkim równoważne — o ile tylko związane wystąpienia a, b, c zmienimy na *pies*, *kot*, *mysz* i nie było wcześniej zmiennych wolnych *pies*, *kot*, *mysz*!

[12] Bo te wszystkie rzeczy można zdefiniować w λ-rachunku — jest to bowiem zupełny język programowania, choć prosty.

5.4. Haskell w implementacji GHC

Jako przykład dojrzałego języka funkcyjnego wybraliśmy **wysokopoziomowy, czysto funkcyjny, silnie typowany, leniwy język ogólnego przeznaczenia** — Haskell[13] [46, 59]. W tym rozdziale będziemy używać jednej z najpopularniejszych jego implementacji, to jest *Glasgow Haskell Compiler* (GHC) w wersji 6.8.2. Haskell ponadto jest językiem modularnym (mimo, że nie jest obiektowy, to mamy hermetyzację w obrębie modułów), polimorficznym (funkcje są ściśle typowane, ale mogą być typowane zbiorami typów, nie tylko konkretnymi typami), kontraktowym (klasy typów wyznaczają kontrakty, czyli interfejsy, dla typów, które mają do nich należeć).

Pliki, które mają być samodzielnie działającymi programami, muszą mieć zdefiniowaną daną[14] `main` typu `IO ()`, która jest *akcją* (więcej na temat tego typu i akcji w Podrozdziale 5.8.1 na stronie 106). Można je wtedy skompilować i uruchamiać jako samodzielne aplikacje.

Jednak większość przykładów tu prezentowanych to zbiory definicji, które nie są w powyższym sensie programami. Możemy je za to łatwo testować w trybie interaktywnym, uruchamiając

```
ghci plik.hs
```

(oczywiście `plik.hs` to plik z testowanymi definicjami; `.hs` to standardowe rozszerzenie plików Haskellowych). Po takim uruchomieniu Haskella zgłosi się znak zachęty `*Main>` (jeśli uruchomimy tryb interaktywny bez pliku, będzie to `Prelude>`) — chyba, że kompilacja nie powiedzie się z powodu błędów, wtedy dostaniemy odpowiedni komunikat. Teraz możemy podawać interpreterowi wyrażenia, a on będzie odpowiadał nam wypisując ich obliczoną wartość (lub komunikat o błędzie; w przypadku akcji będzie powodował jeszcze efekty uboczne).

Przyjrzyjmy się przykładowej konwersacji z GHC (po uruchomieniu `ghci` bez pliku wejściowego), pokazanej na Listingu 5.1.

Listing 5.1. Pierwsze spotkanie z GHC w trybie interaktywnym

```
1 GHCi, version 6.8.2: http://www.haskell.org/ghc/
  :? for help
3 Loading package base ... linking ... done.
  Prelude> 3+4
5 7
  Prelude> (+) 3 4
7 7
```

[13] Język zawdzięcza swą nazwę wybitnemu matematykowi i logikowi, Haskellowi Curry'emu [75]. Istnieje także język Curry, nazwany od jego nazwiska, oraz funkcja `curry` (dostępna także w Haskellu, odwrotna do funkcji `uncurry` wspomnianej na stronie 110, Podrozdział 11).

[14] Wcześniej pisaliśmy o 'funkcji `main`', teraz — o 'danej `main`'. Jak się wkrótce przekonamy, w Haskellu dane to po prostu funkcje bezparametrowe.

```
   Prelude> 7^50
 9 1798465042647412146620280340569649349251249
   Prelude> 7**50
11 1.798465042647412e42
   Prelude> "Ala" ++ "␣ma␣" ++ "kota."
13 "Ala␣ma␣kota."
   Prelude> 'A' : "licja"
15 "Alicja"
   Prelude> pi**1.5
17 5.568327996831708
   Prelude> sin pi
19 1.2246063538223773e-16
   Prelude> sin (pi*0.5)
21 1.0
   Prelude> :t 'A'
23 'A' :: Char
   Prelude> :t "Ala"
25 "Ala" :: [Char]
   Prelude> :t pi
27 pi :: (Floating a) => a
   Prelude> :t (:)
29 (:) :: a -> [a] -> [a]
   Prelude> :t (++)
31 (++) :: [a] -> [a] -> [a]
   Prelude> :t sin
33 sin :: (Floating a) => a -> a
   Prelude> :quit
35 Leaving GHCi.
```

Objaśnienia do Listingu 5.1:

— wiersze 4–7 pokazują zwykłe dodawanie — w notacji infiksowej i prefiksowej (w tej ostatniej nawiasy są konieczne);

— wiersze 8–11 to potęgowanie w dwóch wersjach: dla wykładnika całkowitego `^` i dla dowolnego `**`, ale wtedy wynik jest zmiennoprzecinkowy; przy okazji widać, że liczby całkowite (typu `Integer`) mają nieograniczony zakres;

— wiersze 12–13 to sklejanie napisów; natomiast wiersze 14–15 to doklejanie znaku na początek napisu;

— w wierszach 16–21 widzimy użycie stałej π i funkcji sinus;

— kolejne wiersze przedstawiają specjalne polecenia interpretera, które zaczynają się od dwukropka (nie są to funkcje Haskellowe!) — `:t` podaje typ wyrażenia w postaci Haskellowej deklaracji (z podwójnym dwukropkiem `::`), nawiasy kwadratowe `[]` oznaczają listę, pojedyncza strzałka `->` — funkcję, podwójna strzałka `=>` zostanie omówiona osobno; inne ważne polecenia interpretera to `:quit`, `:?` oraz `:info`.

5.5. Podstawowe przykłady

Rozważmy plik z różnymi definicjami silni przedstawiony na Listingu 5.2 (wszystkie wcięcia są istotne!).

Listing 5.2. Silnia w Haskellu

```
1 silnia1 0 = 1
  silnia1 n = n * silnia1 (n-1)
3
  silnia2 :: Integer -> Integer
5 silnia2 0 = 1
  silnia2 n = n * silnia2 (n-1)
7
  silnia3 n = silniaPOM n 1
9     where silniaPOM 0 x = x
            silniaPOM n x = silniaPOM (n-1) (x*n)
11
  silnia4 :: Integer -> Integer
13 silnia4 n = silniaPOM n 1
      where silniaPOM 0 x = x
15          silniaPOM n x = silniaPOM (n-1) (x*n)
```

W definicjach tych nie mamy żadnych wyrażeń warunkowych, bo zamiast tego mamy *dopasowywanie argumentów do wzorca* — jest kilka definicji funkcji[15] (tutaj po dwie) i w konkretnej aplikacji wybierana jest pierwsza z definicji, która pasuje do konkretnego argumentu. Można definicje warunkowe zapisywać jeszcze inaczej (*dozory* oraz wyrażenie warunkowe — Listing 5.3).

Listing 5.3. Definicje warunkowe inaczej

```
1 silnia1a n
      | n==0       = 1
3     | otherwise  = n * silnia1a (n-1)

5 silnia1b n =
      if n == 0
7         then 1
          else n * silnia1b (n-1)
```

Przyjrzyjmy się teraz konwersacji w trybie interaktywnym po załadowaniu naszych definicji silni (Listing 5.4).

Listing 5.4. Test typów silni

[15] Nazwy wszystkich funkcji, zmiennych, danych muszą zaczynać się w Haskellu od małej litery. Nazwy typów, ich klas i ich konstruktorów — od wielkiej litery.

```
*Main> :t silnia1
silnia1 :: (Num t) => t -> t
*Main> :t silnia2
silnia2 :: Integer -> Integer
*Main> :t silnia3
silnia3 :: (Num a) => a -> a
*Main> :t silnia4
silnia4 :: Integer -> Integer
```

Sprawdzamy tutaj typy naszych funkcji. Jak wcześniej widzieliśmy, podwójny dwukropek `::` oznacza, że dana należy do danego typu (użyliśmy go zresztą w Listingu 5.2), natomiast pojedyncza strzałka `->` oznacza, że dana jest funkcją — po lewej mamy typ parametru, po prawej typ wyniku. Nazwy zaczynające się od dużej litery to nazwy typów lub klas typów, natomiast zaczynające się od małej litery to metazmienne typowe[16].

W przypadku funkcji `silnia2` oraz `silnia4` typ jest taki, jaki podaliśmy. Jednakże Haskell, jako język silnie typowany, chce znać typ każdej danej. Ponieważ jednak nie wymusza jawnego deklarowania typów, musi sam o nich wnioskować z definicji danych, a dokładniej — z typów funkcji, które zostały użyte w definicji. Haskell przyjmuje najszerszy możliwy typ (lub klasę typów) — znalezienie takiego jednego typu (jednej klasy typów) jest zawsze możliwe — chyba, że mamy niezgodność typów, wtedy oczywiście zobaczymy błąd kompilacji lub wykonania.

Tak więc, funkcje `silnia1` oraz `silnia3` mają typ wywnioskowany automatycznie, jego opis wygląda tak[17]:

```
(Num a) => a -> a
```

i należy to odczytać następująco: „dla każdego typu `a` klasy `Num` (czyli liczbowego) funkcja przyjmuje wartość tego typu i wartość tego samego typu wydaje". Jak widać nasze funkcje `silnia1` i `silnia3` są polimorficzne, bo mogą mieć argumenty i wartości wielu typów (byle typów klasy `Num`).

Zobaczmy, jak działają te funkcje, dla różnych argumentów: (Listing 5.5).

Listing 5.5. Test działania silni

```
*Main> silnia1 10
3628800
*Main> silnia2 10
3628800
*Main> silnia3 10
```

[16] Metazmienne, bo nie są to zwykłe zmienne oznaczające wartości pierwszego rzędu, lecz wartości z systemu stojącego wyżej — metasystemu typów. Nie możemy więc w Haskellu definiować funkcji na typach (inaczej niż to jest w Pythonie).

[17] Oczywiście nazwa metazmiennej typowej nie ma znaczenia, raz może być `a`, innym razem `t` albo coś jeszcze innego.

```
6 3628800
  *Main> silnia4 10
8 3628800
  *Main> silnia1 10.5
10 *** Exception: stack overflow
  *Main> silnia2 10.5
12 <interactive>:1:8:
      No instance for (Fractional Integer)
14      arising from the literal '10.5' at
          <interactive>:1:8-11
16    Possible fix: add an instance declaration
        for (Fractional Integer)
18    In the first argument of 'silnia2', namely '10.5'
      In the expression: silnia2 10.5
20    In the definition of 'it': it = silnia2 10.5
  *Main> silnia4 10.5
22 <interactive>:1:8:
      No instance for (Fractional Integer)
24      arising from the literal '10.5' at
          <interactive>:1:8-11
26    Possible fix: add an instance declaration
        for (Fractional Integer)
28    In the first argument of 'silnia4', namely '10.5'
      In the expression: silnia4 10.5
30    In the definition of 'it': it = silnia4 10.5
  *Main> silnia3 10.5
```

Pierwsze wywołania (dla argumentu 10) przynoszą spodziewany wynik. Podobnie jest z wywołaniami `silnia1`, `silnia2`, `silnia4` dla argumentu 10.5 (`silnia1` przepełnia w końcu stos, natomiast `silnia2` i `silnia4` nie pozwalają na taki argument, bo są ograniczone do liczb całkowitych). Co jednak z wywołaniem `silnia3 10.5`? Nigdy się nie skończy[18] — ani poprawnym wynikiem (co oczywiste), ani przepełnieniem stosu, bo stosu nie używa! Dlaczego?

W definicjach funkcji `silnia3` i `silnia4` użyliśmy *rekurencji ogonowej*, bo funkcja zdefiniowana rekurencyjnie[19] występuje w swojej własnej definicji jako funkcja zewnętrzna (wiersze 10 i 15 w Listingu 5.2), a więc jest ostatnim wywołaniem potrzebnym do obliczenia wyniku. Przy takiej konstrukcji (i przy referencyjnej przezroczystości) nie ma potrzeby budowania stosu, bo ramka kolejnego wywołania może **zastąpić** ramkę poprzedniego wywołania zamiast odkładać się na stosie od nowa.

Widać więc, że rekurencja ogonowa zastępuje efektywnie pętle, bo podobnie jak one nie zużywa dodatkowej pamięci z każdym obrotem. Oczywi-

[18] Musimy przerwać pracę GHC, wciskając Ctrl-C.

[19] W obu przypadkach jest ona lokalna względem funkcji głównej, wprowadzona za pomocą `where` i wcięć, a nazywa się `silniaPOM`

ście więc, należy rekurencję ogonową stosować wszędzie tam, gdzie to tylko możliwe.

5.6. Struktury danych i typy

Jednym z atutów Haskella jest system typów. Typowanie tu jest statyczne i bardzo silne, ale polimorficzne. Co więcej typy — jak już widzieliśmy — nie muszą być deklarowane, ale mogą być automatycznie (i statycznie) przez kompilator wywnioskowane z definicji danych i funkcji.

5.6.1. Listy i krotki

Najbardziej podstawową strukturą w językach wysokiego poziomu są listy. W Haskellu definicja list jest klasyczna (Podrozdział 3.3.2.6 na stronie 60), choć używa się nieco innej notacji (elementy rozdzielone przecinkami, w nawiasach kwadratowych) i innych nazw operacji (konstruktor `:` oraz selektory `head` oraz `tail`). Największą różnicą jest jednak konieczność zachowania jednolitego typu wszystkich elementów listy, a wynika to ze ścisłej kontroli typów. W Scheme'ie więc, czy w Lispie, poprawną, 4-elementową, listą jest `'(1 2 (3 4) ((5) (6 7)))`, natomiast w Haskellu nie ma listy `[1, 2, [3, 4], [[5], [6, 7]]]`, ale mogą być listy `[1, 2, 3]`, `[1]`, `[1, 2, 3, 4, 5]`, `[]`, `[[1, 2], [3]]`, `[[[1], [2]], [[3]]]` — pierwsze trzy są tego samego typu, a czwarta nadtypu wszystkich list.

Z drugiej strony mamy w Haskellu krotki (pary, trójki itd.), które mogą mieć różne typy elementów, ale te typy oraz liczba elementów są z góry ustalone. Są więc poprawne w Haskellu krotki `(1, [2, 3])`, czy też `(1, 2, (3, 4), ((5), (6, 7)))`, ale obie są różnych typów.

Spróbujmy napisać kilka prostych operacji listowych — Listing 5.6 (większość ma wbudowane odpowiedniki lub też łatwo je złożyć ze standardowych funkcji).

Listing 5.6. Różne operacje na listach

```
1  sekwencjaRosnaca pocz kon krok
       | pocz > kon    = []
3      | otherwise     = pocz : sekwencjaRosnaca (pocz+krok) kon krok

5  wycinek _    _    []     = []
   wycinek pocz kon lista
7      | pocz > kon         = []
   wycinek 0    kon (g:o)  = g : wycinek 0 (kon-1) o
9  wycinek pocz kon (g:o)  = wycinek (pocz-1) (kon-1) o

11 ponumeruj _     []     = []
   ponumeruj start (g:o)  = (start, g) : ponumeruj (start+1) o
```

```
13
   suma []       = 0
15 suma (g : o) = g + (suma o)

17 iloczyn []       = 1
   iloczyn (g : o) = g * (iloczyn o)
19
   polacz []       = []
21 polacz (g : o) = g ++ (polacz o)

23 parami _         []       = []
   parami []        _        = []
25 parami (g1:o1) (g2:o2)  = (g1, g2) : parami o1 o2

27 redukuj f elNeutralny []       = elNeutralny
   redukuj f elNeutralny (g : o)  = f g (red f elNeutralny o)
29
   mapuj _ []      = []
31 mapuj f (g:o)  = (f g) : mapuj f o

33 filtruj _ []      = []
   filtruj p (g:o)
35     | p g         = g : filtruj p o
       | otherwise   = filtruj p o
```

Definicje powinny być jasne — szczególnie w kontekście testów na Listingu 5.8, więc wyjaśnienia ograniczymy do minimum:

— dwukropek : jest konstruktorem listy (tworzy listę z głowy i ogona), ale może służyć także po lewej stronie definicji do dopasowywania argumentu do wzorca (jak w wierszach 8, 9, 12, 16 i dalej);

— znak podkreślenia _ pełni rolę zmiennej anonimowej — użyty może być wielokrotnie, zawsze oznacza nową daną, która po prawej stronie definicji będzie zignorowana; tak więc w linii 5 dwa podkreślenia oznaczają dwie, być może różne, wartości;

— w definicji funkcji `wycinek`[20] mieszamy swobodnie dozory z dopasowywaniem wzorca.

W definicjach powyższych nie deklarowaliśmy typów, więc sprawdźmy jak o typach wywnioskował sam Haskell (Listing 5.7).

Listing 5.7. Wywnioskowane automatycznie typy funkcji z Listingu 5.6

```
   sekwencjaRosnaca :: (Ord a, Num a) => a -> a -> a -> [a]
2  wycinek :: (Ord a1, Num a1) => a1 -> a1 -> [a] -> [a]
   ponumeruj :: (Num a) => a -> [t] -> [(a, t)]
```

[20] Funkcję `wycinek` można także zdefiniować za pomocą standardowych funkcji `take` oraz `drop`; na przykład następująco:

```
wycinek pocz kon lista = drop pocz (take (kon+1) lista)
```

```
4 suma :: (Num t) => [t] -> t
  iloczyn :: (Num t) => [t] -> t
6 polacz :: [[a]] -> [a]
  parami :: [t1] -> [t] -> [(t1, t)]
8 redukuj :: (t1 -> t -> t) -> t -> [t1] -> t
  mapuj :: (t -> a) -> [t] -> [a]
10 filtruj :: (a -> Bool) -> [a] -> [a]
```

Wszystko powinno być jasne, ewentualnego wyjaśnienia wymagają poniższe kwestie.

— Klasa `Ord`, to typy, których wartości możemy porównywać (bo używamy w dwóch pierwszych definicjach funkcji logicznej `>`).

— `Bool` to typ logiczny.

— Zapis `[y]` oznacza typ list składających się z elementów typu `y`, natomiast `[[x]]` to oczywiście typ list składających się z list składających się z typu `x`. Zapis `(u, v)` to typ krotek składających się z dwóch elementów, pierwszy typu `u`, a drugi typu `v`. W końcu `[(u, v)]` to typ list takich krotek.

— Jak czytać zapisy w rodzaju `a -> b -> c`? Z definicji funkcji można wywnioskować, że ten zapis oznacza funkcję dwuparametrową o typie pierwszego parametru `a`, drugiego — `b`, a wyniku `c`. Jednakże, biorąc pod uwagę to, że `->` jest operatorem, który wiąże prawostronnie, powyższy zapis jest równoważny zapisowi: `a -> (b -> c)`. To jednak znaczy, że jest to funkcja biorąca jeden argument typu `a` i zwracając wartość typu `b -> c`, czyli funkcję! W pewnym sensie obie te interpretacje są uzasadnione, ale formalnie rzecz biorąc **wszystkie funkcje w Haskellu są jednoargumentowe**[21], natomiast mogą zwrócić funkcję, która może zostać zaaplikowana do drugiego argumentu i tak dalej...

— W końcu trzy ostatnie funkcje jako swój pierwszy argument przyjmują inne funkcje.

Działanie tych funkcji pokazuje Listing 5.8.

Listing 5.8. Działanie funkcji z Listingu 5.6 (tryb interaktywny)

```
  *Main> sekwencjaRosnaca 1 10 3
2 [1,4,7,10]
  *Main> sekwencjaRosnaca 1 15 3
4 [1,4,7,10,13]
  *Main> sekwencjaRosnaca (-100) 100 0.1
6 [-100.0,-99.9,-99.80000000000001,-99.70000000000002,
  -- ciach!
8 ,99.7999999999972,99.89999999999719,99.99999999999719]
  *Main> wycinek 80 83 (sekwencjaRosnaca (-100) 100 0.5)
10 [-60.0,-59.5,-59.0,-58.5]
```

[21] Poza funkcjami bezargumentowymi, które są utożsamiane ze stałymi.

```
   *Main> ponumeruj 100 "Curry"
12 [(100,'C'),(101,'u'),(102,'r'),(103,'r'),(104,'y')]
   *Main> suma (sekwencjaRosnaca 1 100 1)
14 5050
   *Main> iloczyn (sekwencjaRosnaca 1 10 1)
16 3628800
   *Main> polacz ["Ala", "ska"]
18 "Alaska"
   *Main> parami "Dama" "Kier"
20 [('D','K'),('a','i'),('m','e'),('a','r')]
   *Main> redukuj (+) 0 [1,2,3,4,5,6]
22 21
   *Main> redukuj (*) 1 [1,2,3,4,5,6]
24 720
   *Main> redukuj (++) "" ["Ala", "ska"]
26 "Alaska"
   *Main> redukuj (++) [] ["Ala", "ska"]
28 "Alaska"
   *Main> mapuj (2^) (sekwencjaRosnaca 0 10 1)
30 [1,2,4,8,16,32,64,128,256,512,1024]
   *Main> filtruj ((== '1') . head . show)
32                (mapuj (2^) (sekwencjaRosnaca 0 20 1))
   [1,16,128,1024,16384,131072,1048576]
```

Co tu się dzieje? Jedyne niejasności mogą być w następujących miejscach:

— wiersz 7 tak naprawdę zastępuje wyciętych kilkaset wierszy z wynikowej listy, bo ta jest strasznie długa;

— wiersze 11–12, 17–20, 25–28 dowodzą, że napis to rzeczywiście lista znaków;

— notacja `(operator argument)`[22] oraz `(argument operator)` (wiersze 29, 31, 32) to *sekcje* — funkcje z ustalonym drugim lub pierwszym parametrem — więc równoważne są następujące zapisy:

```
6/3
(/) 6 3
(6/) 3
(/3) 6
```

— wiersze 31–32 powinny być zapisane w jednej linii.

Funkcja `mapuj`[23] aplikuje swój pierwszy argument (który ma być funkcją) do każdego z elementów listy i zwraca listę wyników. To, co się dzieje w wierszach 29–30 jest więc oczywiste.

[22] Ta pierwsza forma nie działa dla odejmowania, bo `(-x)` oznacza po prostu liczbę o przeciwnym znaku niż `x`. Tu przy okazji warto też wspomnieć, że w Haskellu negacja liczby musi być zapisywana w nawiasie (jak w wierszu 5 czy 9) — nigdy `-x`.

[23] Jest analogiczna funkcja standardowa `map`.

Funkcja `filtruj`[24] sprawdza każdy z elementów danej listy z użyciem funkcji logicznej[25] danej w pierwszym parametrze i w wyniku pozostawia tylko te spełniające ów predykat. Czytelnik sam wydedukuje, co dzieje się w wierszach 31–33[26].

Warto zauważyć tutaj, że funkcje `suma`, `iloczyn`, `polacz` mają ten sam, bardzo popularny, schemat operowania na liście danych[27] — różnią się tylko wykonywaną operacją oraz elementem początkowym odpowiednim dla danej operacji (neutralnym). Dlatego też możemy pokusić się o zdefiniowanie funkcji uniwersalnej `redukuj`[28], która realizuje ten schemat, a jako parametry przyjmuje ową operację i element neutralny (a także, oczywiście, listę). Możemy teraz zapisać nowe definicje funkcji `suma`, `iloczyn`, `polacz` (Listing 5.9).

Listing 5.9. Nowe definicje za pomocą `redukuj`

```
1 suma1 lista = redukuj (+) 0 lista
  iloczyn1 lista = redukuj (*) 1 lista
3 polacz1 lista = redukuj (++) [] lista
  suma2 = redukuj (+) 0
5 iloczyn2 = redukuj (*) 1
  polacz2 = redukuj (++) []
```

Wymowa pierwszych trzech wierszy Listingu 5.9 powinna być całkiem jasna. Natomiast wiersz 4–6 równoważne[29] są wierszom 1–3, a korzystają z tego, że funkcje są w Haskellu wartościami pierwszego rzędu (czyli mogą być używane, ale także definiowane jak dane) i z tego, że każda funkcja jest właściwie jednoargumentowa (strona 94), więc gdy wywołamy ją z liczbą argumentów mniejszą niż maksymalna, wtedy po prostu dostaniemy w wyniku nową funkcję z ustalonym jednym lub więcej parametrami[30]. Tak więc argumenty „wiszące" po obu stronach definicji można bezpiecznie opuszczać.

[24] I tu mamy analogiczną funkcję predefiniowaną `filter`.

[25] Czyli: *predykatu*.

[26] Biorąc pod uwagę, że `head` zwraca pierwszy element listy, `show` zwraca argument w postaci napisu, a kropka `.` jest operatorem składania funkcji (czyli `f(g(h x))`) to to samo, co `(f . g . h) x`.

[27] Taka operacja nazywa się właśnie *redukcją*

[28] Funkcje standardowe `foldr`, `foldl` i pokrewne realizują podobne operacje.

[29] Ze względu na pewną kontrowersyjną zasadę stosowaną w Haskellu, funkcje te mogą nie być równoważne co do typów domyślnych (druga trójka będzie miała typy zawężone w stosunku do pierwszej trójki, która będzie miała typy takie jak oryginalne funkcje `suma`, `iloczyn`, `polacz` z Listingu 5.6). Możemy jednak zawsze typy zdefiniować samemu.

[30] Jest to tak zwane *domknięcie*, którego pewną odmianą są wspomniane na stronie 95 sekcje.

5.6.1.1. Zapisy skrótowe i listy składane

Listy są bardzo powszechnie stosowaną w Haskellu strukturą danych, więc język ten dysponuje kilkoma przydatnymi skrótami, pokazuje je Listing 5.10. Listing ten pokazuje też zapis zwany *listami składanymi* (ang. *list comprehension*) i porównanie ich z zapisem wykorzystującym funkcje standardowe `map` oraz `filter`, a także zapis funkcji anonimowych w postaci λ-wyrażeń (patrz też Podrozdział 5.7). Zwróćmy uwagę, że zapisy z linii 13 oraz 22 przypominają wiernie następujące zapisy matematyczne[31]: $\{2^x | x \in \{1,\ldots,10\}\}$ oraz $\{(x^2, 2^y) | x \in \{1,\ldots,4\}, y \in \{1,\ldots,3\}\}$.

Listing 5.10. Skrótowe zapisy list i listy składane

```
   Prelude> [1..10]
2  [1,2,3,4,5,6,7,8,9,10]
   Prelude> ['a'..'k']
4  "abcdefghijk"
   Prelude> [1,4..10]
6  [1,4,7,10]
   Prelude> [1,4..15]
8  [1,4,7,10,13]
   Prelude> ['a','d'..'k']
10 "adgj"
   Prelude> [1,3..]
12 [1,3,5,7,9,11,13,15, -- ciach! (bo lista nieskonczona)
   Prelude> [2^x | x <- [1..10]]
14 [2,4,8,16,32,64,128,256,512,1024]
   Prelude> map (\x -> 2^x) [1..10]
16 [2,4,8,16,32,64,128,256,512,1024]
   Prelude> [2^x | x <- [1..15], '1' == head (show (2^x))]
18 [16,128,1024,16384]
   Prelude> map (\x->2^x)
20           (filter (\x -> '1' == head (show (2^x))) [1..15])
   [16,128,1024,16384]
22 Prelude> [(x^2, 2^y) | x <- [1..4], y <- [1..3]]
   [(1,2),(1,4),(1,8),(4,2),(4,4),(4,8),(9,2),(9,4),(9,8),
24  (16,2),(16,4),(16,8)]
```

5.6.2. Własne typy

Jak w każdym języku wysokiego poziomu, w Haskellu możemy definiować własne typy, także skomplikowane typy strukturalne.

[31] Choć w polskich publikacjach częściej używa się dwukropka w miejsce kreski pionowej.

5.6.2.1. Rekurencyjne struktury danych

Podstawową rekurencyjną strukturą danych jest lista, która jest wbudowana w język. My spróbujemy zdefiniować inny prosty typ rekurencyjny — drzewo binarne — wraz z pewnymi na nim operacjami (Listing 5.11).

Listing 5.11. Drzewo binarne w Haskellu

```
data DBin typ = DPuste
              | DPelne typ (DBin typ) (DBin typ)
              deriving (Show, Eq)

czyDPuste DPuste           = True
czyDPuste (DPelne _ _ _)   = False

-- trawersowanie drzewa sposobem in-order
drzewoWListe DPuste  = []
drzewoWListe (DPelne korzen lewe prawe)
    = drzewoWListe lewe ++ korzen : drzewoWListe prawe

-- wstawianie do drzewa poszukiwan
wstawDoBST co DPuste  = DPelne co DPuste DPuste
wstawDoBST co (DPelne x lewe prawe)
    | co > x          = DPelne x lewe (wstawDoBST co prawe)
    | otherwise       = DPelne x (wstawDoBST co lewe) prawe

listaWBST []     = DPuste
listaWBST (g:o)  = wstawDoBST g (listaWBST o)

przykladoweDrzewo = DPelne 5
                           (DPelne 1 DPuste DPuste)
                           przykladoweDrzewo
d1 = DPelne 1
            DPuste
            (DPelne 2
                    (DPelne 3 DPuste DPuste)
                    (DPelne 4 DPuste DPuste))
d2 = DPelne 5 d1 (DPelne 6 DPuste DPuste)
d3 = DPelne 7 d1 d2
```

Definicja właściwego typu zawarta jest w wierszach 1–3. Ustalamy tu jego nazwę `DBin` oraz parametr `typ`[32]. Dane naszego typu mogą występować w dwóch formach, dla których ustalamy konstruktor: bezparametrowy `DPuste` oraz trzyparametrowy `DPelne`. Oba te konstruktory pełnią rolę funkcji, co można sprawdzić w trybie interaktywnym (Listing 5.12).

[32] Bo nasz typ drzew binarnych jest polimorficzny (parametryczny) i jest w rzeczywistości zbiorem typów — każdy z typów jest dookreślony typem składowych `typ`. Gdybyśmy chcieli definiować tylko typ drzew liczb całkowitych, to zamiast parametru pojawić się tu winno `Integer`. Ale po co się ograniczać, skoro można zdefiniować typ polimorficzny?

W końcu, w linii 3 ustalamy, że nasz typ ma być zaliczony do klas typów `Show` (która pozwala dostawać daną w formie napisu za pomocą funkcji `show`) oraz `Eq` (która pozwala na stosowanie operatorów równości `==` oraz różności `/=`). Przy tym każemy Haskellowi automatycznie zaimplementować potrzebne funkcje (`show`, `==`, `/=`).

Ciąg dalszy to definicje funkcji, które powinny być dla uważnego czytelnika jasne. Jeszcze tylko test działania naszego drzewa (Listing 5.12).

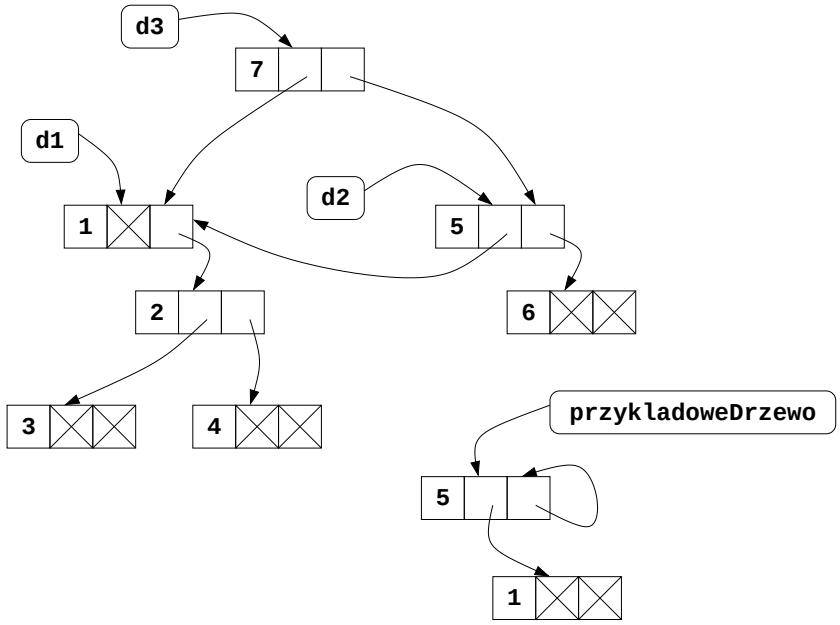
Listing 5.12. Test drzewa

```
1  *Main> :t DPuste
   DPuste :: DBin typ
3  *Main> :t DPelne
   DPelne :: typ -> DBin typ -> DBin typ -> DBin typ
5  *Main> :t czyDPuste
   czyDPuste :: DBin t -> Bool
7  *Main> :t drzewoWListe
   drzewoWListe :: DBin a -> [a]
9  *Main> :t wstawDoBST
   wstawDoBST :: (Ord typ) => typ -> DBin typ -> DBin typ
11 *Main> :t listaWBST
   listaWBST :: (Ord typ) => [typ] -> DBin typ
13 *Main> :t przykladoweDrzewo
   przykladoweDrzewo :: DBin Integer
15 *Main> listaWBST [1,5,2,7,5,2,234,78,53,32,34,64,75]
   DPelne 75 (DPelne 64 (DPelne 34 (DPelne 32 -- ciach!
17 *Main> drzewoWListe (
            listaWBST [1,5,2,7,5,2,234,78,53,32,34,64,75])
19 [1,2,2,5,5,7,32,34,53,64,75,78,234]
   *Main> listaWBST [1,5,2] == listaWBST [1,2,5]
21 False
   *Main> listaWBST [1,5,2] == listaWBST [5,2,1]
23 False
   *Main> listaWBST [1,5,2] == listaWBST [5,1,2]
25 True
   *Main> drzewoWListe przykladoweDrzewo
27 [1,5,1,5,1,5,1,5,1,5,1,5,1,5,1,5,1, -- ciach!
```

Złożone struktury danych są też okazją do pokazania trwałości danych. Rozważmy dane `przykladoweDrzewo`, `d1`, `d2`, `d3` zdefiniowane jak w Listingu 5.11. Schemat ich ulokowania w pamięci operacyjnej pokazuje Rysunek 5.1. Właściwie zastosowane jest tu aliasowanie, ale ze względu na przezroczystość referencyjną jest ono bezpieczne.

Wygląda to właśnie tak, bo w językach czysto funkcyjnych mamy dane, które nie mogą być zmienione[33]. W związku z tym, dane mogą współdzielić te same elementy i jest tak, jeśli wynika to z definicji. Dzięki temu oszczędza-

[33] Każda zmiana jest tutaj zawsze utworzeniem nowej danej.



Rysunek 5.1. Reprezentacja w pamięci (nieco uproszczona) drzew `przykladoweDrzewo`, `d1`, `d2`, `d3` z Listingu 5.11

my czas tworzenia danych (bo nie musimy wielu rzeczy kopiować), a także pamięć. Ma to swoje wady (brak transformatorów), wspominaliśmy o nich na stronie 84 (Podrozdział 5.2).

5.6.2.2. Struktury nieskończone

Wynik ostatniego wyrażenia (`drzewoWListe przykladoweDrzewo`) z Listingu 5.12 (ale wcześniej pokazuje podobne rzeczy także Listing 5.10) jest nieco zaskakujący — jest to lista nieskończona. Podobnie byłoby z wynikiem `sekwencjaRosnaca 1 10 (-1)` (funkcja z Listingu 5.6). W Haskellu bowiem możemy w podobny sposób definiować struktury nieskończone.

Jakie mogą mieć zastosowanie? Na przykład chcemy znaleźć przybliżoną wartość $\sqrt{a}$ metodą Newtona-Raphsona [58]. Metoda ta — jak wiele metod numerycznych — polega na generacji ciągu kolejnych przybliżeń (który zwykle jest zdefiniowany rekurencyjnie) i wybraniu z tego ciągu elementu, który spełnia jakieś warunki dobrego przybliżenia. Dzięki liście nieskończonej możemy rozdzielić to zagadnienie na dwa niezależne problemy [60] — generację ciągu i wyłuskiwanie odpowiedniego przybliżenia.

Najpierw zajmijmy się rekurencyjnym ciągiem nieskończonym (Listing 5.13). Zależność między kolejnymi wyrazami tego ciągu, to

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}. \tag{5.5}$$

Listing 5.13. Algorytm Newtona-Raphsona, część I

```
ciagRekurencyjnyNR a
    = ciagRekurencyjny a (zaleznosc a)
          where zaleznosc s x = 0.5 * (x+s/x)
                ciagRekurencyjny start f
                  = start : ciagRekurencyjny (f start) f
```

Teraz kolej na wybór dobrego przybliżenia (Listing 5.14). Wybieramy taki wyraz ciągu, który od poprzedniego różni się mniej niż o zadane ε.

Listing 5.14. Algorytm Newtona-Raphsona, część II

```
dobraAproksymacja epsilon (x1:x2:reszta)
    | abs(x1-x2) < epsilon
      = x2
    | otherwise
      = dobraAproksymacja epsilon (x2:reszta)
```

I przykładowe połączenie (Listing 5.15).

Listing 5.15. Algorytm Newtona-Raphsona, część III

```
mojPierwiastek a = dobraAproksymacja
            (0.00000000001*a) (ciagRekurencyjnyNR a)
```

Taki podział na podzadania niezależne ma kilka zalet:

— dzięki leniwemu obliczaniu takie funkcje mogą działać współbieżnie/równolegle, w modelu *producenta-konsumenta* — `ciagRekurencyjnyNR` produkuje wartości ciągu — ale leniwie: nie wszystkie (szczególnie, że to niemożliwe, bo ciąg jest nieskończony), tylko tyle, ile potrzebuje ich konsument czyli `dobraAproksymacja`;

— osobne zadania mogą być programowane przez różnych programistów, a ponieważ mamy do czynienia z przezroczystością referencyjną, to nie ma żadnego problemu efektów ubocznych, ważna jest jedynie dobra specyfikacja wejścia (argumentów) dla konsumenta i wyjścia (wyników) dla producenta;

— łatwo zmienić każdą z funkcji niezależnie od drugiej — kiedy chcemy inaczej znajdować wystarczającą aproksymację, zmieniamy funkcję

`dobraAproksymacja`, a jak chcemy rozwiązać inny problem numeryczny, zmieniamy funkcje `ciagRekurencyjnyNR` (gdzie może na przykład wystarczyć tylko zmiana lokalnej funkcji `zaleznosc`).

5.6.2.3. Bezpieczne unie

W podobny do drzewa (patrz wyżej, Podrozdział 5.6.2.1), ale prostszy sposób możemy zdefiniować bezpieczną unię (tutaj dwóch elementów, Listingi 5.16–5.17).

Listing 5.16. Unie w Haskellu wraz z przykładowymi danymi i funkcjami

```
data Unia typ1 typ2 = Jedno typ1 | Drugie typ2
    deriving(Eq, Show)

-- funkcja testowa
wybierzJedno [] = []
wybierzJedno ((Jedno x):o) = x : wybierzJedno o
wybierzJedno ((Drugie x):o) = wybierzJedno o

-- dane testowe
listaZnakowILiczb = [Jedno 'k', Drugie 13, Jedno 'o',
                     Drugie 100, Drugie 5]

unijnaListaLiczb  = map Drugie [1, 2, 3]
unijnaListaZnakow = map Jedno "ala"

razem = listaZnakowILiczb
      ++ unijnaListaLiczb
      ++ unijnaListaZnakow

zwierz = wybierzJedno razem
```

Listing 5.17. Działanie naszej unii w Haskellu

```
*Main> :t Jedno
Jedno :: typ1 -> Unia typ1 typ2
*Main> :t Drugie
Drugie :: typ2 -> Unia typ1 typ2
*Main> :t wybierzJedno
wybierzJedno :: [Unia a t] -> [a]
*Main> :t listaZnakowILiczb
listaZnakowILiczb :: [Unia Char Integer]
*Main> :t unijnaListaLiczb
unijnaListaLiczb :: [Unia typ1 Integer]
*Main> :t unijnaListaZnakow
unijnaListaZnakow :: [Unia Char typ2]
*Main> :t razem
razem :: [Unia Char Integer]
*Main> razem
```

```
16 [Jedno 'k',Drugie 13,Jedno 'o',Drugie 100,Drugie 5,
   Drugie 1,Drugie 2,Drugie 3,Jedno 'a',Jedno 'l',Jedno 'a']
18 *Main> :t zwierz
   zwierz :: [Char]
20 *Main> zwierz
   "koala"
```

Unia ta jest bezpieczna (w przeciwieństwie do rozważanych w Podrozdziale 3.3.2.2 na stronie 49), bo jest konstrukcją wysokopoziomową, przechowującą nie tylko daną, ale także jej typ (jak to w językach silnie typowanych) oraz jednoznaczny indykator wskazujący, który element unii mamy na uwadze (albo `Jedno`, albo `Drugie`).

5.7. Rachunek lambda w Haskellu

Wrócimy tu jeszcze do implementacji λ-rachunku w Haskellu. Jak zapewne się Czytelnik zorientował, zapis `a b` jest w Haskellu zapisem aplikacji — bo oznacza obliczenie funkcji `a` na argumencie `b`. Jak zapisujemy abstrakcję? Otóż zapis $\lambda x.y$ ma w Haskellu odpowiednik `\x->y` (bo podobno znak `\` przypomina nieco literę λ). Listing 5.18 pokazuje podstawowe operacje λ-rachunku na przykładach w Haskellu.

Listing 5.18. Proste przykłady użycia λ-rachunku

```
1 *Main> (\x -> 2 + x) 4
  6
3 *Main> :t (\x -> 2 + x)
  (\x -> 2 + x) :: (Num t) => t -> t
5 *Main> (\f -> 2 + f 4) sin
  1.2431975046920718
7 *Main> :t (\f -> 2 + f 4)
  (\f -> 2 + f 4) :: (Num t1, Num t) => (t1 -> t) -> t
9 *Main> (\f -> 2 + f 4) (\x -> 2 + x)
  8
11 *Main> map (\x -> 2*x**3 - 4*x**2 + 7*x - 12) [1,-3,10,-12]
   [-7.0,-123.0,1658.0,-4128.0]
```

Kilka klasycznych definicji λ-rachunku [7] przedstawiają poniższe wzory[34].

$$TRUE \quad = \quad \lambda x.\lambda y.x \tag{5.6}$$

[34] Te (i inne, bardziej skomplikowane) przykłady mogą stanowić pewne potwierdzenie, że λ-rachunek jest zupełnym językiem programowania, w którym można wyrazić wszelkie obliczenia.

$$FALSE = \lambda x.\lambda y.y \tag{5.7}$$
$$IFTHENELSE = \lambda p.\lambda a.\lambda b.pab \tag{5.8}$$
$$PAIR = \lambda x.\lambda y.\lambda f.fxy \tag{5.9}$$
$$FIRST = \lambda p.pTRUE \tag{5.10}$$
$$SECOND = \lambda p.pFALSE \tag{5.11}$$

Spróbujmy je zaimplementować w Haskellu (Listing 5.19) i wypróbować ich działanie (Listing 5.20).

Listing 5.19. Klasyczne definicje lambda rachunku w Haskellu

```
lTRUE      = \ x -> \ y -> x
lFALSE     = \ x -> \ y -> y
lIFTHENELSE = \ p -> \ a -> \ b -> p a b
lPAIR      = \ x -> \ y -> \ f -> f x y
lFIRST     = \ p -> p lTRUE
lSECOND    = \ p -> p lFALSE
```

Listing 5.20. Klasyczne definicje w działaniu

```
*Main> lIFTHENELSE lTRUE "tak" "nie"
"tak"
*Main> lIFTHENELSE lFALSE "tak" "nie"
"nie"
*Main> lIFTHENELSE lFIRST (lPAIR lTRUE lFALSE) "tak" "nie"
"tak"
*Main> lFIRST (lPAIR 1 2)
1
*Main> lSECOND (lPAIR 1 2)
2
```

5.8. Monady

Na koniec pozostał problem efektów ubocznych, pamięci zewnętrznej, interakcji z użytkownikiem, wyjątków, rozległych struktur danych z dostępem swobodnym, transformatorów danych... Jednym słowem: rzeczy, bez których nie wyobrażamy sobie dzisiaj programowania, a które — jak na razie — nie wyglądają na zgodne z programowaniem funkcyjnym.

Dla ustalenia uwagi skupimy się na problemie konwersacji z użytkownikiem, ale rozwiązania wymienionych wyżej problemów nie różnią się od tego w głównych założeniach.

Chcemy więc napisać program, który pyta użytkownika o imię i wita się z nim. W Pythonie (patrz też Rozdział 7) wyglądać mógłby on tak, jak na Listingu 5.21 lub na Listingu 5.22.

Listing 5.21. Dwulinijkowe powitanie użytkownika w Pythonie

```
imie = raw_input("Jak␣sie␣nazywasz?␣")
print "Witaj,␣" + imie + "!"
```

Listing 5.22. Jednolinijkowe powitanie użytkownika w Pythonie

```
print "Witaj,␣" + raw_input("Jak␣sie␣nazywasz?␣") + "!"
```

Niestety, funkcja `raw_input` daje różne wyniki dla tego samego argumentu `"Jak sie nazywasz? "`[35], nie jest więc funkcją w sensie programowania czysto funkcyjnego[36].

Drugi problem to lenistwo i dowolność w kolejności wykonywania obliczeń języków funkcyjnych a potrzebna nam tutaj *gorliwość*gorliwa ewaluacja (czyli wykonywanie obliczeń od razu, jak tylko się pojawią) i ustalona kolejność obliczeń języków imperatywnych.

Niektóre języki funkcyjne (Lisp, Scheme) rezygnowały z czystości funkcyjnej na rzecz udostępnienia funkcji nieczystych (jak `raw_input` w Pythonie), przy czym niektóre z języków wyraźnie starały się oddzielić funkcje czyste od nieczystych (choćby przez ich nazwę lub specjalny typ). Inne (Clean, Mercury) implementują *typ!unikalnego występowania* (ang. *uniqueness type*), co oznacza mniej więcej, że w owym typie jest zmienna unikalna, mogąca mieć wiele wartości, oznaczających kolejne stany na przykład świata zewnętrznego, ale tylko jedna wartość — bieżąca — dostępna jest w owym typie w danej chwili.

W Haskellu używane są do tego celu (i do podobnych) *monady*. Jest to pojęcie wzięte z teorii kategorii, działu matematyki którym zajmować się nie będziemy. Wystarczy nam wiedzieć, że monada reprezentuje sekwencyjne wykonywanie pewnych *akcji*.

Monada jest każdym zbiorem typów klasy `Monad`, wymuszającej zdefiniowanie trzech operacji (patrz też Listing 5.23):

— `return x`, która konstruuje trywialną akcję zwracającą `x`;

— `f >>= g`, która składa sekwencyjnie dwie akcje w jedną, polegającą na wykonaniu **najpierw** akcji `f`, a potem akcji `g` **na wyniku akcji** `f`[37];

[35] Sprawdzaliśmy. Gdy jedno z nas siedzi przy komputerze wynikiem jest napis `"Jarek"`, gdy drugie — napis `"Beata"`.

[36] Co jest równoważne temu, że nie jest po prostu funkcją w sensie matematycznym.

[37] Co odpowiada składaniu funkcji w programowaniu imperatywnym, jak w Listingu 5.22.

— `f >> g`, która składa sekwencyjnie dwie akcje w jedną, polegającą na wykonaniu **najpierw** akcji `f`, a potem akcji `g` **bez uwzględniania wyniku akcji** `f`[38].

W obu powyższych złożeniach, bez względu na to, czy akcja `g` zależy, czy nie, od wyniku akcji `f`, **może ona zależeć od pozostawionych przez nią efektów ubocznych**, które wewnętrzna implementacja monady może „po kryjomu" przekazywać dalej wzdłuż operacji `>>=` oraz `>>=`.

Listing 5.23. Typy operacji monadowych

```
1 return :: (Monad m) => a -> m a
  (>>=)  :: (Monad m) => m a -> (a -> m b) -> m b
3 (>>)   :: (Monad m) => m a -> m b -> m b
```

5.8.1. Wejście/wyjście

Dzięki monadom można w wygodny — a do tego czysty funkcyjnie i osadzony w systemie ścisłej kontroli typów — sposób opisywać miedzy innymi wejście/wyjście programu, w tym interakcję z użytkownikiem. Pozwalają one także na ścisłą separację części funkcyjnej (która jest zwykle łatwiejsza do pielęgnacji i utrzymania poprawności) od części monadowej („imperatywnej").

Trzeba podkreślić, że wartościami typu monadowego są **akcje, nie ich wyniki**. Kiedy więc wykonywane są owe akcje i kiedy pojawiają się ich wyniki? Znowu obowiązuje tu zasada lenistwa. Formalnie bowiem wynikami funkcji monadowych są akcje **opisujące wszystkie możliwe drogi obliczeń**, ale z lenistwa żadna z nich nie jest obierana do ostatniego momentu, czyli aż do rzeczywistego wykonania programu. Wtedy to dopiero konkretna droga obliczeń jest wybierana na podstawie danych z zewnątrz.

Formalnie programem w Haskellu jest dana o nazwie `main` i typie `IO ()` (czyli: `main :: IO ()`), opisującą pewną (złożoną) akcję. Jak widać, nie jest to prawdziwa funkcja, lecz **stała** (nie ma w deklaracji symbolu funkcji `->`), a więc opis akcji wykonywanych przez program jest niezależny od żadnych parametrów — choć już samo wykonanie akcji (w czasie działania programu) może zależeć na przykład od tego co z zewnątrz (na wejściu) zostało dostarczone.

Co oznacza typ `IO ()`? `IO` jest monadą obsługującą wejście/wyjście. Każdy typ monadowy musi być skonkretyzowany typem zwracanym przez daną akcję, a ponieważ program jest zamkniętą całością, to nie zwraca nic,

[38] Co odpowiada sekwencji instrukcji w programowaniu imperatywnym.

bo się kończy. Owo 'nic' należy do *typu jednostkowego* (ang. *unit type*)[39], którego jest jedyną wartością[40]. Zarówno typ jednostkowy, jak i jego jedyną wartość 'nic' oznaczamy `()`. Normalne funkcje czyste, które dają wynik typu jednostkowego nie są zbyt ciekawe, bo zgodnie z przezroczystością referencyjną są sobie wszystkie równoważne. Jednak w programowaniu akcji może się to przydać, bo dwie akcje o wartości typu pustego mogą różnić się efektami ubocznymi.

Omówmy kilka przykładowych standardowych akcji i spróbujmy złożyć je w program witający się po imieniu z użytkownikiem.

Listing 5.24. Przykładowe akcje standardowe

```
1 getLine  :: IO String
  putStrLn :: String -> IO ()
3 putStr   :: String -> IO ()
```

Widzimy (Listing 5.24), że typ `getLine` wskazuje na to, że jest to akcja bezparametrowa (bo nie jest funkcją), ale zwraca napis. Z kolei dwie pozostałe przyjmują jakiś parametr napisowy i od niego zależy ich działanie (bo mamy funkcję `->`), ale nic nie zwracają (typ wyniku akcji jest jednostkowy)[41].

Możemy w trybie interaktywnym uruchamiać akcje, zatem spróbujmy (Listing 5.25). Działają zgodnie z oczekiwaniem, przy `getLine` dodatkowo widzimy, że tryb interaktywny Haskella uruchamia akcje (wyświetlając przy tym ich wyniki i realizując inne operacje, jak wpisanie `Haskell` przez użytkownika w wierszu 5), nie ograniczając się jedynie do ich wyznaczenia. Widać też, że składanie działa zgodnie z oczekiwaniem: w pierwszym złożeniu (wiersz 7) używamy `>>=`, bo potrzebujemy przekazać wynik z pierwszej akcji do drugiej, a w drugim (wiersz 10) `>>`, bo nie potrzebujemy; w trzecim złożeniu (wiersz 13) mamy oba operatory.

Listing 5.25. Test akcji

```
1 Prelude> putStr "Ala ma kota."
  Ala ma kota.Prelude> putStrLn "Ala ma kota."
3 Ala ma kota.
  Prelude> getLine
5 Haskell
  "Haskell"
7 Prelude> getLine >>= putStrLn
```

[39] Nie należy mylić go z *typem pustym* (ang. *null type*), który nie ma wartości i oznaczać może jedynie błąd!

[40] Jest tu pewne podobieństwo do `void` z języka C, czy też `None` z Pythona (strona 142, Podrozdział 7.3.1.1).

[41] A raczej: zwracają 'nic'.

```
Napis
Napis
Prelude> putStr "Jak␣sie␣nazywasz:␣" >> getLine
Jak sie nazywasz: Eustachy
"Eustachy"
Prelude> putStr "Jak␣sie␣nazywasz:␣" >> getLine >>= putStrLn
Jak sie nazywasz: Eustachy
Eustachy
```

Niestety, nie możemy napisać czegoś takiego jak w Listingu 5.26, bowiem mieszamy wtedy akcje z funkcjami — próbujemy traktować akcje jako zwykłe funkcje i składać je z innymi funkcjami. Ścisła kontrola typów na to nie pozwala!

Listing 5.26. Złe składanie akcji

```
Prelude> putStrLn ("Witaj,␣"
                   ++ (putStr "Jak␣sie␣nazywasz:␣" >> getLine)
                   ++ "!")
<interactive>:1:24:
    Couldn't match expected type `[a]' against inferred
                                                  type `IO ()'
    In the first argument of `(>>)', namely
        `putStr "Jak␣sie␣nazywasz:␣"'
    In the first argument of `(++)', namely
        `(putStr "Jak␣sie␣nazywasz:␣" >> getLine)'
    In the second argument of `(++)', namely
        `(putStr "Jak␣sie␣nazywasz:␣" >> getLine) ++ "!"'
```

Musimy to zrobić inaczej. Listing 5.27 pokazuje cztery **równoważne** definicje funkcji `main` realizującej zadanie zaimplementowane w Pythonie na Listingach 5.21 oraz 5.22. Omówimy te definicje krótko poniżej.

Listing 5.27. Dobre składanie akcji z użyciem funkcji

```
main1 = putStr "Jak␣sie␣nazywasz:␣"
        >> getLine
        >>= powitaj
            where powitaj imie
                    = putStrLn ("Witaj,␣" ++ imie ++ "!")

main2 = putStr "Jak␣sie␣nazywasz:␣"
        >> getLine
        >>= (\imie -> putStrLn ("Witaj,␣" ++ imie ++ "!"))

main3 = putStr "Jak␣sie␣nazywasz:␣"
        >> getLine
        >>= \imie -> putStrLn ("Witaj,␣" ++ imie ++ "!")

```

```
main4 = putStr "Jak␣sie␣nazywasz:␣"
        >> getLine >>= \imie
        -> putStrLn ("Witaj,␣" ++ imie ++ "!")

main5 = do putStr "Jak␣sie␣nazywasz:␣"
           imie <- getLine
           putStrLn ("Witaj,␣" ++ imie ++ "!")
```

W `main1` rozwiązujemy problem przez zdefiniowanie własnej funkcji `powitaj` zależnej od parametru i zwracającej akcję używającą tego parametru. Jednakże, takie podejście ma tę wadę, że w bardziej skomplikowanym programie musielibyśmy zdefiniować wiele takich funkcji pomocniczych. Na szczęście wiemy, że dzięki λ-rachunkowi (Podrozdział 5.7) możemy wstawiać w potrzebne miejsce od razu funkcje anonimowe, co wykorzystujemy w definicja `main2`.

W definicji `main3` usuwamy w stosunku do `main2` tylko nawiasy, bo nie są tu konieczne. Zaś definicja `main4` jest dokładnie taka, jak `main3`, różni się jedynie podziałem na wiersze — zwracamy tu uwagę na to, że zmienna pomocnicza `imie` dostaje wynik z akcji `getLine`.

W końcu ostatnia definicja — naprawdę równoważna poprzednim — używa notacji `do`, która jest cukrem składniowym dla składania akcji w sposób bardziej przypominający programowanie imperatywne i ograniczającym natłok i różnorodność (w `main2`–`4` mamy ich już wszystkie trzy rodzaje: `>>`, `>>=`, `->`) strzałek.

Na koniec uwaga: `<-` przypomina podstawienie, ale nim nie jest. Zapis z tą strzałką oznacza uruchomienie akcji, wyciągnięcie jej wyniku i skojarzenie go ze zmienną, która jest lokalna względem dalszej części i która — jak wszystkie dane w językach funkcyjnych — już swojej wartości nie zmieni. Może jednak zostać przysłonięta, bo jeśli pojawi się w jednym bloku `do` zapis: `x <- ... ... x <- ...`, to drugie `x` przysłania pierwsze, a nie jest tym samym (dokładnie tak jak w λ-rachunku, którego elementów pewnym zapisem jest notacja `do`).

5.9. Pytania i zadania

1. Wyjaśnij pojęcia: λ-rachunek, λ-abstrakcja, α-konwersja, β-redukcja, aplikacja, czystość funkcyjna, przezroczystość referencyjna, trwałe struktury danych, leniwa ewaluacja, gorliwa ewaluacja, wartości pierwszego rzędu, rekurencja ogonowa, monada, akcja, dozór, sekcja, domknięcie, lista składana, typ jednostkowy, typ pusty, typ unikalnego występowania, konstruktor, transformator, selektor.

2. Zdefiniuj w Haskellu typ `Pracownik`, którego wartości mogłyby przechowywać dane o pracowniku, takie jak: imię, nazwisko, datę urodzenia, stanowisko. Zdefiniuj odpowiednie selektory do wyłuskiwania odpowiednich elementów.
3. Jak można zdefiniować algorytmy wymagające powtarzania nie mając do dyspozycji pętli?
4. Jakie są zalety leniwego obliczania?
5. Jakie zastosowania mają struktury nieskończone?
6. Dlaczego zdefiniowane przez nas Unie są w Haskellu bezpieczne? Na czym to bezpieczeństwo polega?
7. Zdefiniuj w Haskellu dowolną funkcję, przyjmującą jeden argument (daną prostą), a dającą w wyniku funkcję. Jaki jest jej typ?
8. Zapisz funkcję anonimową, która dla danej pary (krotki dwuelementowej) zwraca parę tych samych elementów, ale w odwrotnej kolejności.
9. Zapisz funkcję anonimową przyjmującą dwa argumenty (funkcyjne), a zwracającą ich złożenie.
10. Korzystając z β-redukcji uprość wyrażenia: $(\lambda x.x)(\lambda x.x)$, $(\lambda x.x)(\lambda y.y)z$, $(\lambda x.\lambda y.xy)z$, $(\lambda x.xx)(\lambda x.xx)$.
11. Czym są `f` oraz `t` zdefiniowane poniżej?

```
f = 0 : map (uncurry (+)) (zip f (1:f))
t = map (uncurry (+)) (zip [0..] (0:t))
```

Funkcja standardowa `uncurry` mogłaby być zdefiniowana następująco:

```
uncurry f (x,y) = f x y
```

natomiast `map` oraz `zip` to standardowe funkcje odpowiadające funkcjom `mapuj` oraz `parami` z Listingu 5.6 (stron 92).

12. Stosując schemat z Podrozdziału 5.6.2.2 zdefiniuj funkcję `miejsceZerBis f a b epsilon`, która znajduje metodą bisekcji miejsce zerowe funkcji rzeczywistej `f :: Double -> Double` w przedziale $\langle$`a`;`b`$\rangle$ z dokładnością `epsilon` (to jest: rozwiązanie otrzymane musi się różnić od prawdziwego o mniej niż `epsilon`).
13. Spróbuj zdefiniować w Haskellu własną implementację list na wzór znanych z Lispa (Podrozdział 3.3.2.6 na stronie 60) oraz zawartej tutaj (strona 98, Podrozdział 5.11) definicji drzew. Konstruktorami typu mają być `ListaPusta` oraz `Para`, a dodatkowo zdefiniuj funkcje `glowa`, `ogon`, `czyPusta`.
14. * Zdefiniuj dla własnych list (zdefiniowanych w poprzednim zadaniu) funkcje: `polacz l1 l2` (zwraca sklejenie list `l1` i `l2`), `zawiera x l` (zwraca prawdę gdy `x` jest elementem listy `l`), `odwroc l` (zwraca listę `l` od końca).

15. * Zdefiniuj funkcję, która usuwa dany element z binarnego drzewa poszukiwań (strona 98, Podrozdział 5.11), zwracając drzewo bez danego elementu, ale z zachowanym uporządkowaniem pozostałych.
16. * Zdefiniuj typ, który pozwala przechowywać listy Lispowe — czyli takie, których elementami mogą być zarówno elementy proste, jak i listy (Lispowe) tych elementów (Podrozdział 3.3.2.6 na stronie 60).
17. Jakie zalety ma rozdzielenie części funkcyjnej programu od imperatywnej?
18. Jak można pogodzić czystość funkcyjną z funkcjami losowymi lub operacjami wejścia/wyjścia?
19. Jakie problemy powstają w programowaniu funkcyjnym przy implementacji struktur o dostępie swobodnym?
20. Co oznaczają w Haskellu poniższe zapisy?

```
a :: t
b :: t -> u
c :: t -> u -> v
d :: (t -> u) -> v
e :: t -> (u -> v)
f :: IO ()
g :: IO t
h :: t -> IO ()
i :: t -> IO u
```

21. * Napisz w Haskellu program — czyli funkcję `main :: IO ()` — który pyta użytkownika o dwie liczby rzeczywiste i w odpowiedzi wyświetla ich sumę.

Rozdział 6

Programowanie logiczne

6.1. Budowa programu logicznego

Idea programowania logicznego rozumianego jako automatyczne wnioskowanie na podstawie przesłanek sięga lat 50. XX wieku (*advice taker* [35] zaproponowany przez wspomnianego w poprzednim rozdziale Johna McCarthy'ego). Jednakże, takiej postaci, z jaką mamy do czynienia dzisiaj, programowanie logiczne nabrało na początku lat 70. XX wieku [24, 25, 26]. Wtedy powstały teoretyczne podstawy algorytmu mechanicznego wnioskowania — SLD-rezolucji (patrz niżej, Podrozdział 6.3) i pierwsze implementacje Prologa, pierwszego języka opartego na SLD-rezolucji. Kolejnym ważnym krokiem w implementowaniu paradygmatu logicznego stało się opracowanie *abstrakcyjnej maszyny Warrena* (WAM) [3, 64] — modelu logicznej maszyny wirtualnej, która stała się faktycznym standardem dla Prologa i umożliwiła jego prekompilację, a więc poprawiła efektywność działania.

Program w języku logicznym [42] składa się z dwóch części:

— *baza wiedzy*, którą stanowi lista *klauzul*, o których zakłada się, że są prawdziwe;

— *zapytanie*, które jest klauzulą do sprawdzenia/udowodnienia przez program — na podstawie powyższej bazy wiedzy.

Plik wejściowy dla języka logicznego może być też samą bazą wiedzy, bez określonego zapytania. Wtedy można potraktować go analogicznie do bibliotek w innych językach — jako zestaw definicji do wykorzystania w różnych programach.

Klauzula jest alternatywą prostych wyrażeń logicznych, z których wszystkie, żadne lub niektóre mogą być zanegowane[1]. Ogólnie można klauzulę zapisać następująco:

$$P_1 \vee \ldots \vee P_m \vee (\neg Q_1) \vee \ldots \vee (\neg Q_n), \tag{6.1}$$

a równoważnie (korzystając z praw De Morgana):

$$(P_1 \vee \ldots \vee P_m) \vee (\neg(Q_1 \wedge \ldots \wedge Q_n)), \tag{6.2}$$

i w końcu (korzystając z prawa $(p \Rightarrow q) \Leftrightarrow ((\neg p) \vee q)$)[2]:

$$(P_1 \vee \ldots \vee P_m) \Leftarrow (Q_1 \wedge \ldots \wedge Q_n). \tag{6.3}$$

W programowaniu logicznym używa się jedynie *klauzul Horna*, w których może być maksymalnie jedno niezanegowane wyrażenie (więc, w zapisach

[1] Wszystkie, żadne lub niektóre mogą być również niezanegowane. Klauzula może być pusta, bez żadnych wyrażeń składowych — taka klauzula ma wartość logiczną 'fałsz' z definicji.

[2] Nie bez powodu zapisujemy w (6.3) i dalej implikację skierowaną w lewo — taki zapis przyjęty jest w języku Prolog (i chyba jest bardziej „programistyczny", jak się okaże poniżej).

(6.1)–(6.3): $m \in 0, 1, n \in \mathbb{N}$), zwane *głową* klauzuli (reszta zwana jest jej *ciałem*). W związku z tym możemy wyróżnić trzy rodzaje klauzul Horna (poniżej wszędzie $n \geqslant 1$):

$$P_1 \Leftarrow , \tag{6.4}$$

$$P_1 \Leftarrow (Q_1 \wedge \ldots \wedge Q_n), \tag{6.5}$$

$$\Leftarrow (Q_1 \wedge \ldots \wedge Q_n). \tag{6.6}$$

Korzystając z wywodu (6.1)–(6.3) możemy zapisać (6.4) oraz (6.6) następująco:

$$P_1, \tag{6.7}$$

$$(\neg Q_1) \vee \ldots \vee (\neg Q_n). \tag{6.8}$$

Klauzule z samą głową (6.4)/(6.7) nazywane są *faktami* i można je czytać: „bezwarunkowo zachodzi P_1". Klauzule mające głowę i ciało, czyli postaci (6.5) nazywane są *zależnościami* (także: *regułami*) i można je czytać: „żeby zachodziło P_1, wystarczy, żeby zachodziło Q_1 oraz ... oraz Q_n". Te dwa rodzaje klauzul składają się na wspomnianą na początku bazę wiedzy.

Natomiast klauzula bez głowy (6.6)/(6.8) jest wspomnianym wyżej *zapytaniem* i można interpretować ją jako pytanie „czy i kiedy spełnione jest Q_1 oraz ... oraz Q_n?".

Czym są natomiast $P_1, Q_1, \ldots, Q_n$ — owe „proste wyrażenia logiczne", o których wspomnieliśmy na samym początku? Są to formuły postaci $p(t_1, \ldots, t_k)$, $k \geqslant 0$, przy czym p jest nazwą *predykatu* (czyli funkcji, która przyjmuje wartości logiczne), natomiast jego argumenty $t_1, \ldots, t_k$ są *termami*. Termy mogą być z kolei trojakiego rodzaju (definicja jest rekurencyjna):
— stałe,
— zmienne,
— struktury postaci $f(t_1, \ldots, t_l)$, $l \geqslant 0$[3], gdzie f jest nazwą *funktora* budującego strukturę, a $t_1, \ldots, t_l$ są termami.

Celem funktorów (a także stałych) jest opisywanie obiektów z którymi pracuje program, natomiast celem predykatów — opisywanie relacji między tymi obiektami[4]. Zmienne natomiast traktuje się jak w zasięgu kwantyfikatora ogólnego ($\bigwedge$)[5].

[3] Strukturę bezargumentową ($l = 0$) utożsamiamy ze stałą f.

[4] Predykaty mogą występować w roli funktorów, bo języki logiczne — jak Prolog — są refleksyjne.

[5] W związku z tym zmienne są lokalne względem swojej klauzuli. Ponadto nie zachowują się jak zmienne w programowaniu imperatywnym: w programowaniu logicznym, jeśli zmienna dostanie jakąś wartość, to już tej wartości nie może zmienić, dopóki rozpatrywana jest ciągle jedna klauzula i jeden jej przypadek.

6.2. Język Prolog

Język Prolog (w różnych odmianach) jest najpopularniejszym językiem programowania logicznego, a inne języki logiczne są mniej lub bardziej na Prologu oparte. Klauzule w Prologu zapisuje się niemal dokładnie jak we wzorach (6.4)–(6.6), jedyne różnice, to:
— każda klauzula kończy się kropką;
— znak implikacji $\Leftarrow$ zastąpiony jest symbolem `:-` (dwukropek, minus);
— znak koniunkcji w klauzuli zastąpiony jest przecinkiem;
— fakty zapisywane są bez znaku implikacji.
Ponadto obowiązują pewne konwencje:
— wszystkie nazwy zwykłych zmiennych rozpoczynają się od dużej litery (i zawierają jedynie cyfry, duże i małe litery alfabetu angielskiego i znaki pokreślenia);
— wszystkie nazwy stałych, funktorów, predykatów powinny być podawane w pojedynczych cudzysłowach (w apostrofach); jeśli jednak rozpoczynają się od małej litery, a zawierają jedynie cyfry, duże i małe litery alfabetu angielskiego i znaki pokreślenia, to apostrofy można (a dla czytelności należy) opuścić;
— istnieje zmienna anonimowa `_` (znak pokreślenia, jak w Haskellu), która w każdym swoim wystąpieniu jest formalnie osobnym obiektem i jej ewentualna wartość jest po ustaleniu ignorowana/zapominana — zmienna ta służy jako wypełniacz;
— pewne zapisy mają specjalne znaczenie: liczby całkowite i zmiennoprzecinkowe są traktowane jako specjalne stałe; zapis listowy jest specjalnym funktorem (Podrozdział 6.4), a napisy (w podwójnych cudzysłowach) są tożsame z listami kodów znaków.
Trzeba jeszcze zaznaczyć, że zarówno predykatów, jak i funktorów nie deklarujemy w żaden inny sposób niż przez ich wystąpienie w klauzulach. Mogą być także przeciążane.

6.2.1. Praca z SWI-Prologiem

Konkretna implementacja Prologa, którą posługujemy się w tym rozdziale, to SWI-Prolog (wersja 5.6.58). Najwygodniej testować podane tu przykłady (oraz własne próbki) zapisując je jako zwykły plik tekstowy, a potem uruchamiając interpreter w trybie interaktywnym z załadowanym owym plikiem[6]:

[6] Rozszerzenie nazwy dla plików Prologowych związane jest z pewnymi niejednoznacznościami. SWI-Prolog przyjmuje za domyślne `.pl`, ale jak wiadomo jest to też rozszerzenie Perla. Używa się więc innych, na przykład `.pro`, `.prolog` — rozszerzenie nie ma jednak znaczenia przy ładowaniu w sposób, który tu podaliśmy.

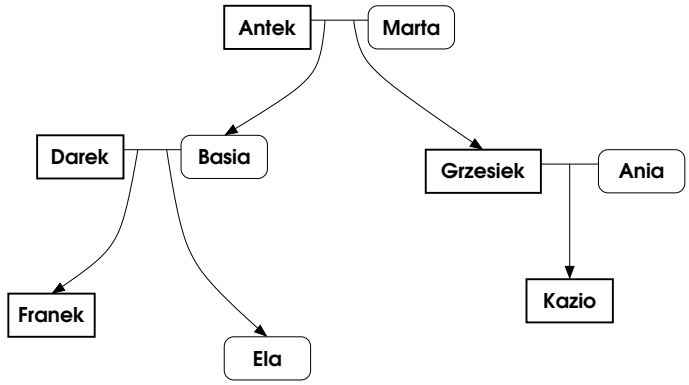
```
swipl -s plik.pl
```

Po załadowaniu zgłasza się interpreter w oczekiwaniu na zapytanie (w podanych przykładach pliki nie zawierają zapytań, zadajemy je w trybie interaktywnym, po poniższym symbolu, który zastępuje implikację):

```
?-
```

które wpisujemy po owym znaku zachęty bez wiodącego znaku implikacji (porównaj (6.6)). Mogą być wielolinijkowe, zawsze muszą być zakończone kropką!

Obejrzyjmy pierwszy z programów, implementujący zależności rodzinne z Rysunku 6.1 — Listing 6.1[7].



Rysunek 6.1. Przykładowe drzewo genealogiczne

Listing 6.1. Przykładowa rodzina w Prologu

```
1  rodzic(antek, grzesiek).
   rodzic(antek, basia).
3  rodzic(marta, grzesiek).
   rodzic(marta, basia).
5  rodzic(grzesiek, kazio).
   rodzic(ania, kazio).
7  rodzic(darek, franek).
   rodzic(darek, ela).
9  rodzic(basia, franek).
   rodzic(basia, ela).
11
   kobieta(marta).
```

[7] Będziemy trzymać się konwencji, że predykaty nazywamy czasownikami oznaczającymi relację lub orzecznikami tych relacji, pierwszy argument predykatu będzie podmiotem zdania, a kolejne — dopełnieniami. Tak więc `rodzic(basia, ela)` oznacza 'Basia jest rodzicem Eli'.

```
13 kobieta(basia).
   kobieta(ania).
15 kobieta(ela).

17 matka(X, Y) :- rodzic(X, Y), kobieta(X).

19 corka(X, Y) :- rodzic(Y, X), kobieta(X).

21 babcia(X, Y) :- matka(X, Z), rodzic(Z, Y).

23 rodzenstwo(X, Y) :- rodzic(Z, X), rodzic(Z, Y), X \= Y.

25 siostra(X, Y) :- rodzenstwo(X, Y), kobieta(X).
```

Listing 6.2. Przykładowa rodzina w Prologu — test interaktywny

```
1 ?- rodzic(antek, kazio).
  false.
3 ?- rodzic(antek, basia).
  true.
5 ?- matka(Matka, grzesiek).
  Matka = marta ;
7 false.
  ?- matka(basia, Dziecko).
9 Dziecko = franek ;
  Dziecko = ela.
11 ?- matka(antek, Dziecko).
   false.
13 ?- rodzic(Rodzic, grzesiek).
   Rodzic = antek ;
15 Rodzic = marta ;
   false.
17 ?- siostra(S, X).
   S = basia,
19 X = grzesiek ;
   S = basia,
21 X = grzesiek ;
   S = ela,
23 X = franek ;
   S = ela,
25 X = franek ;
   false.
```

Po załadowaniu pliku z Listingu 6.1 możemy spytać o różne rzeczy (Listing 6.2).

— Wiersz 1: czy Antek jest rodzicem Kazia?
— Wiersz 3: czy Antek jest rodzicem Basi?
— Wiersz 5: kto jest matką Grześka?

Tutaj w odpowiedzi Prologa (wiersz 6) nie ma kropki na końcu, a kursor stoi po słowie `marta`, nie widać też znaku zachęty... Prolog jednak nie zawiesił się, lecz daje znak, że nie sprawdził jeszcze wszystkich możliwości i czeka na naszą dalszą decyzję. Możemy wcisnąć kropkę lub Enter, co oznacza, że dalsze rozwiązania nas nie interesują (wracamy wtedy do znaku zachęty), albo wcisnąć średnik lub spację (jak w naszym przykładzie), co oznacza prośbę o poszukanie dalszych możliwości — w naszym przypadku odpowiedzią komputera będzie `false.` (wiersz 7), co oznacza, że dalszych rozwiązań nie ma[8].

— Wiersz 8: czyją matką jest Basia?
Średnik pochodzi tu oczywiście od użytkownika, bo chcemy znaleźć dalsze rozwiązania.
— Wiersz 11: czyją matką jest Antek?
— Wiersz 13: kto jest rodzicem Grześka?
Końcowe `false.` mówi oczywiście o tym, że nie ma **dalszych rozwiązań** oprócz Antka i Marty — kazaliśmy średnikiem szukać dalszych rozwiązań.
— Wiersz 17: kto jest czyją siostrą?

Dzięki temu, że w wierszu 23 Listingu 6.1 dopisaliśmy warunek `X \= Y` nie dostaliśmy odpowiedzi, że Basia jest swoją siostrą, oraz że Ela jest swoją siostrą — bowiem predykat predefiniowany `\=` (infiksowy) daje prawdę, gdy jego argumenty są różne. Odpowiedzi, które tu dostajemy są powtórzone, bo rodzeństwo może być po matce i po ojcu, a tu mamy do czynienia z pełnymi rodzinami. W sensie teorii mnogości nie psuje to rozwiązania, bo powtórzenie elementów w zbiorze nie zmienia jego zawartości.

6.3. Przebieg wnioskowania

Żeby lepiej zrozumieć, jak Prolog dochodzi do rozwiązań — i dlaczego akurat takich, w takiej kolejności — prześledźmy przebieg *SLD-rezolucji*[9] zapytania:

$$\Leftarrow siostra(S, X). \tag{6.9}$$

Zapis ten jest równoważny zapisowi (porównaj (6.6) oraz (6.8)):

$$\neg \bigvee_{S,X} siostra(S, X), \tag{6.10}$$

[8] Nic dziwnego, zwykle ma się jedną matkę...

[9] Rezolucja (ściślej: SLD-rezolucja) to opisany tutaj sposób mechanicznego dowodzenia twierdzeń stosowany w programowaniu logicznym. Przypomina to dowodzenie twierdzeń nie wprost.

a więc zakłada się tu, że nie istnieje pozytywna odpowiedź na dane pytanie. Naszym zadaniem jest znalezienie takich wartości zmiennych S oraz X, dla których założenie to będzie sprzecznością. W związku z tym szukamy w bazie wiedzy (liście klauzul) klauzuli, której głowa może zostać *uzgodniona* z naszym *celem* ($siostra(S, X)$). Wiersz 25 zawiera taką klauzulę[10]:

$$siostra(X_1, Y_1) \Leftarrow rodzenstwo(X_1, Y_1), kobieta(X_1). \tag{6.11}$$

Możemy tutaj dokonać uzgodnienia

$$X_1 = S, Y_1 = X \tag{6.12}$$

i po tym uzgodnieniu zająć się nowym celem:

$$\Leftarrow rodzenstwo(S, X), kobieta(S). \tag{6.13}$$

Teraz celem jest koniunkcja dwóch formuł, żeby znaleźć jej rozwiązanie, musimy kolejno (od lewej do prawej) udowodnić składowe. Tak więc zaczniemy od $rodzenstwo(S, X)$. Wiersz 23 zawiera klauzulę:

$$rodzenstwo(X_2, Y_2) \Leftarrow rodzic(Z_2, X_2), rodzic(Z_2, Y_2), X_2 \neq Y_2. \tag{6.14}$$

Kolejne uzgodnienie, którego trzeba dokonać, to

$$X_2 = S, Y_2 = X \tag{6.15}$$

i teraz zająć się kolejnym celem

$$\Leftarrow rodzic(Z_2, S), rodzic(Z_2, X), S \neq X, \tag{6.16}$$

który znowu składa się z kilku podcelów. Znowu wybieramy pierwszy z nich ($rodzic(Z_2, X)$) i szukamy pierwszej klauzuli, której głowa pasuje do naszego podcelu. Tym razem jest to fakt z wiersza 1:

$$rodzic(antek, grzesiek) \Leftarrow, \tag{6.17}$$

więc pozostaje uzgodnić

$$Z_2 = antek, S = grzesiek, \tag{6.18}$$

[10] W kolejnych krokach wywodu, nazwy zmiennych, które mogą kolidować ze sobą, będziemy modyfikować przez dodawanie indeksów. Jak już wspomniano, zasięg zmiennej w programowaniu logicznym obejmuje dokładnie jedną klauzulę — od momentu wprowadzenia owej zmiennej do kropki. W różnych klauzulach (a także w różnych „podejściach" do sprawdzenia tej samej klauzuli) te same nazwy oznaczają więc inne zmienne!

a ponieważ ostatnia klauzula nie ma ciała, to uznajemy, że doszliśmy do końca wnioskowania — w tej gałęzi drzewa! Teraz musimy z tymi uzgodnieniami wrócić do (6.16) i zająć się drugim podcelem ($rodzic(Z_2, X)$), który, po dokonanych uzgodnieniach jest równoważny: $rodzic(antek, X)$. Pierwsza klauzula pasująca do podcelu to znowu wiersz 1, a uzgodnienie to $X = grzesiek$. pozostaje w (6.16) jeszcze jeden podcel: $S \neq X$, który dla naszego uzgodnienia ($S = grzesiek, X = grzesiek$) jest fałszywy!

Pierwsza próba znalezienia rozwiązania nie udała się. Co teraz? Trzeba wykonać *nawrót*, czyli wrócić do ostatniego momentu, w którym są jeszcze jakieś inne pasujące do któregoś podcelu klauzule i powtórzyć próbę uzgodnienia z tymi właśnie nowymi klauzulami. W naszym przypadku będzie to powrót do (6.16), do drugiego z podcelów i poszukanie kolejnej klauzuli zgadzającej się z $rodzic(antek, X)$. Tym razem będzie to klauzula z wiersza 2: $rodzic(antek, basia)$, więc teraz $X = basia$. Trzeci podcel z (6.16) jest tym razem spełniony (bo $grzesiek \neq basia$), więc z tym uzgodnieniem wracamy do (6.13), do podcelu drugiego ($kobieta(S)$). Niestety[11], nie możemy z głową żadnej klauzuli uzgodnić formuły $kobieta(grzesiek)$ — znowu porażka.

Musimy teraz powrócić jeszcze wcześniej, do momentu, kiedy coś zostało jeszcze do sprawdzenia. Tym razem wracamy do pierwszego podcelu w (6.16), i szukamy kolejnej klauzuli o głowie uzgadnialnej z $rodzic(Z_2, S)$. Teraz kolej na wiersz 2, $rodzic(antek, basia)$, co daje nam uzgodnienie $Z_2 = antek, S = basia$. Drugi podcel w (6.16) (po uzgodnieniu: $rodzic(antek, X)$) uzgadnia się z wierszem 1 (dając $X = grzesiek$) i teraz zarówno ostatni podcel w (6.16) ($S \neq X$), jak i ostatni w (6.13) ($kobieta(S)$) kończy się z sukcesem. Wracając teraz do (6.9) dostajemy pierwsze rozwiązanie $siostra(basia, grzesiek)$. Taką odpowiedź daje też jako pierwszą Prolog, a — jak już wiemy — następnych może poszukać na nasze żądanie.

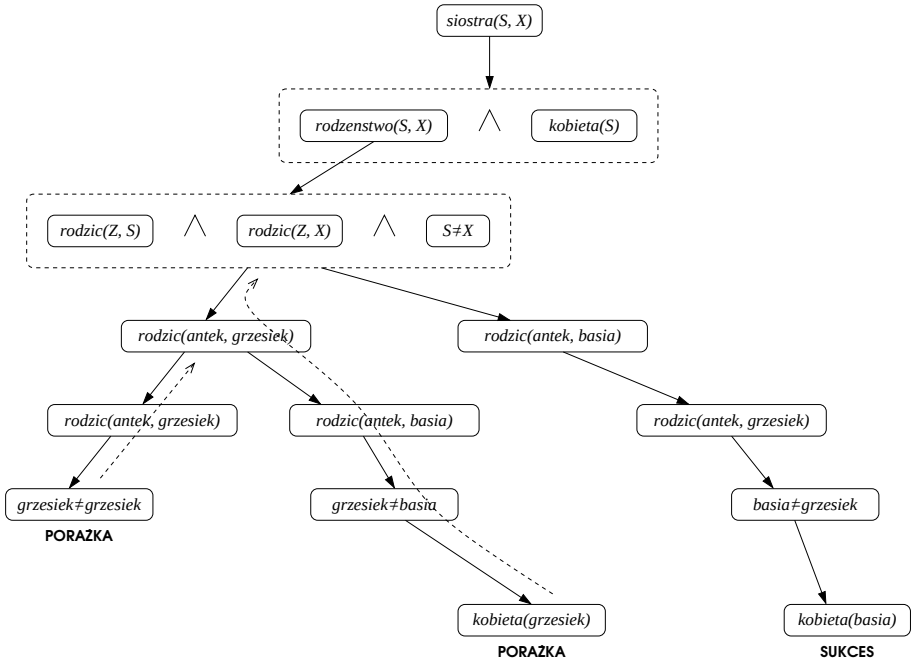
Rysunek 6.2 przedstawia opisany powyżej fragment SLD-rezolucji w postaci *SLD-drzewa*.

Opisane tutaj wnioskowanie można też prześledzić wywołując nasze zapytanie w trybie śledzenia, do którego przenosi nas wywołanie predykatu `trace`. Kolejne kroki przechodzimy (`creep`) wciskając Enter — Listing 6.3

Listing 6.3. Śledzenie

```
?- trace.
Unknown message: query(yes)
[trace]  ?- siostra(S, X).
   Call: (7) siostra(_G312, _G313) ? creep
   Call: (8) rodzenstwo(_G312, _G313) ? creep
   Call: (9) rodzic(_L212, _G312) ? creep
   Exit: (9) rodzic(antek, grzesiek) ? creep
   Call: (9) rodzic(antek, _G313) ? creep
```

[11] A może na szczęście...



Rysunek 6.2. SLD-drzewo przykładowej SLD-rezolucji

```
      Exit: (9) rodzic(antek, grzesiek) ? creep
10    Call: (9) grzesiek\=grzesiek ? creep
      Fail: (9) grzesiek\=grzesiek ? creep
12    Redo: (9) rodzic(antek, _G313) ? creep
      Exit: (9) rodzic(antek, basia) ? creep
14    Call: (9) grzesiek\=basia ? creep
      Exit: (9) grzesiek\=basia ? creep
16    Exit: (8) rodzenstwo(grzesiek, basia) ? creep
      Call: (8) kobieta(grzesiek) ? creep
18    Fail: (8) kobieta(grzesiek) ? creep
      Redo: (9) rodzic(_L212, _G312) ? creep
20    Exit: (9) rodzic(antek, basia) ? creep
      Call: (9) rodzic(antek, _G313) ? creep
22    Exit: (9) rodzic(antek, grzesiek) ? creep
      Call: (9) basia\=grzesiek ? creep
24    Exit: (9) basia\=grzesiek ? creep
      Exit: (8) rodzenstwo(basia, grzesiek) ? creep
26    Call: (8) kobieta(basia) ? creep
      Exit: (8) kobieta(basia) ? creep
28    Exit: (7) siostra(basia, grzesiek) ? creep
  S = basia,
30 X = grzesiek
```

6.3.1. Znaczenie kolejności

Jak widać z powyższego wywodu, kolejność klauzul oraz kolejność formuł w klauzulach ma znaczenie, mimo, że z logicznego punktu widzenia (przemienność koniunkcji w ciele klauzuli, nieuporządkowanie klauzul w zbiorze klauzul) nie powinno to mieć znaczenia.

W powyższym przykładzie (i w dalszych także) wyniki będą te same po zamianie kolejności klauzul (co najwyżej ich kolejność także się zmieni), ale ze względu na dopuszczalne w Prologu efekty uboczne[12], może tak nie być. Kolejność może też mieć wpływ na efektywność znalezienia pierwszego rozwiązania — jeśli kolejność nakazuje przeszukiwanie najpierw fałszywych gałęzi drzewa, szukanie będzie trwało dłużej.

W końcu, niektóre predykaty wymagają argumentów już uzgodnionych, a więc nie można użyć ich w ciele klauzuli zbyt wcześnie. W naszym przykładzie dotyczy to predykatu `\=`, który potrafi porównać tylko dane konkretne, a nie niewiadome. Tak więc przesunięcie w linii 23 formuły `X \= Y` ku przodowi klauzuli spowoduje błędy — dociekliwy Czytelnik zapewne sam to sprawdzi.

6.3.2. Unifikacja

Ważnym pojęciem w programowaniu logicznym jest uzgadnianie, czyli *unifikacja*, o czym pisaliśmy w opisie przykładowej SLD-rezolucji. Dwa wyrażenia da się uzgodnić, jeśli istnieje takie podstawienie do zmiennych występujących w tych wyrażeniach, żeby te wyrażenia były identyczne. Rozważmy dwa wrażenia, które chcemy uzgodnić:

$$f(X, a) \quad \text{oraz} \quad f(Y, Z). \tag{6.19}$$

Wszystkie poniższe koniunkcje równości uzgadniają wyrażenia (6.19):

$$X = a, Y = a, Z = a, \tag{6.20}$$
$$X = b, Y = b, Z = a, \tag{6.21}$$
$$X = Y, Z = a, \tag{6.22}$$
$$X = s(t), Y = s(t), Z = a, \tag{6.23}$$

[12] Jak choćby *cięcie* (zapisywane w Prologu: `!`) i inne predykaty, o których nie będziemy tu mówić. Warto jednak tu wspomnieć, że w przeciwieństwie do języków funkcyjnych, w językach logicznych efekty uboczne są traktowane nieco bardziej przyjaźnie i pewne predykaty mają je po prostu (jak: `write`, `read`, `nl`, `assert`, `consult`, wspomniane wyżej cięcie i wiele innych). Efekty uboczne nie psują bowiem logiczności wywodu, choć mogą wpływać na jego wyniki; psują natomiast przezroczystość referencyjną w językach funkcyjnych. No i zmieniać mogą przyszłe uzgodnienia (ale nie te już dokonane!), więc i wyniki wywodu.

jednakże najlepsze jest tylko jedno, mianowicie (6.22). Dlaczego jest ono najlepsze? Bo jest najogólniejsze[13] z wszystkich uzgodnień dopuszczalnych. Takie więc uzgodnienie nie ogranicza w żaden sposób przyszłych uzgodnień. Takie też uzgodnienia są zawsze wybierane w procesie SLD-rezolucji.

Przyjrzyjmy się jeszcze poniższej konwersacji z Prologiem w trybie interaktywnym (Listing 6.4). Pokazuje ona dokonane unifikacje różnych termów. Do tych eksperymentów wykorzystujemy predykat wbudowany `=`, który daje prawdę, jeśli jego argumenty dadzą się uzgodnić (przeciwieństwo predykatu `\=`) i uzgadnia je.

Listing 6.4. Przykłady uzgodnień

```
?- X = Y.
X = Y.

?- X = Y, Y = a.
X = a,
Y = a.

?- X = f(a), Y = f(X).
X = f(a),
Y = f(f(a)).

?- f(X, a) = f(Y, Z).
X = Y,
Z = a.

?- f(a, f(b, f(c, f(d)))) = f(_, f(_, X)).
X = f(c, f(d)).

?- 2+3*8 = X+Y.
X = 2,
Y = 3*8.

?- 2+X*7 = Y+2*Z.
X = 2,
Y = 2,
Z = 7.

?- 2+3 = 5.
false.

?- X = f(X).
X = f(**).

?- X = f(X).
X = f(**).
```

[13] Takie uzgodnienie zawsze istnieje i jest jedyne (modulo nazwy zmiennych), dowód formalny pozostawimy jednak dociekliwym [42].

```
36
   ?- X = f(X, X).
38 X = f(**, **).

40 ?- X = f(X, Y).
   X = f(**, Y).
42
   ?- X = f(X, Y), Y = g(X).
44 X = f(**, g(**)),
   Y = g(f(**, **)).
```

W piątym zapytaniu (wiersz 16) użyliśmy zmiennej anonimowej `_`. W szóstym, siódmym i ósmym (wiersze 19, 23, 28) natomiast użyliśmy wbudowanych operatorów arytmetycznych, które, jak widać nie służą bezpośrednio obliczeniom, lecz budowaniu struktury wyrażeń[14]. W końcu ostatnich pięć zapytań (wiersze 31–45) definiuje struktury potencjalnie nieskończone — symbol `**` (przypominający nieco wyglądem ∞) wskazuje, że w jego miejscu powinno zostać powtórzone rekurencyjnie coś, co już w tym samym zapisie się pojawiło.

6.4. Listy i struktury

Skoro Prolog potrafi definiować skomplikowane struktury, to spróbujmy zdefiniować operacje listowe na wzór tych z Lispa (Podrozdział 3.3.2.6 na stronie 60).

Chcemy zdefiniować stałą `nil` (oznaczającą listę pustą) oraz funktor dwuargumentowy `cons` (konstruktor pary). Definicje takie w prologu realizuje się „operacyjnie", to jest przez zdefiniowanie predykatów działających na nich. A więc zechcemy zdefiniować predykaty `czyLista(Rzecz)`, `listaPusta(Lista)`, `jestGlowa(Glowa, Lista)`, `jestOgonem(Ogon, Lista)` — Listing 6.5.

Listing 6.5. Przykładowa realizacja list w Prologu

```
1 czyLista(nil).
  czyLista(cons(_, X)) :- czyLista(X).
3
  listaPusta(nil).
5
  jestGlowa(Glowa, cons(Glowa, _)).
7
  jestOgonem(Ogon, cons(_, Ogon)).
```

[14] Prolog pozwala także na definiowanie swoich, dowolnych operatorów, z dowolnymi priorytetami i łącznością [84].

Bardzo to proste, jedyna nowość — czy też pewna komplikacja — to rekurencyjna definicja w wierszu 2. Bo zgodnie z regułami podanymi na stronie 60 (Podrozdział 3.3.2.6) ogon listy też musi być listą.

Ale czy to działa? Pokazuje to Listing 6.6.

Listing 6.6. Test własnej realizacji list

```
  ?- czyLista(1).
2 false.

4 ?- czyLista(nil).
  true.
6
  ?- czyLista(cons(1, cons(2, nil))).
8 true.

10 ?- czyLista(cons(1, cons(2, 3))).
   false.
12
   ?- jestGlowa(X, cons(1, cons(2, nil))).
14 X = 1.

16 ?- jestOgonem(X, cons(1, cons(2, nil))).
   X = cons(2, nil).
18
   ?- jestGlowa(1, L), jestOgonem(nil, L).
20 L = cons(1, nil).

22 ?- jestGlowa(1, L1), jestOgonem(nil, L1),
         jestGlowa(2, L2), jestOgonem(L1, L2).
24 L1 = cons(1, nil),
   L2 = cons(2, cons(1, nil)).
```

Działa, co więcej predykaty `jestGlowa` oraz `jestOgonem` działają „w obie strony" (wiersze 19–25).

Lista jest na szczęście tak podstawową strukturą w językach deklaratywnych, że nie musimy własnoręcznie jej definiować. Istnieje wbudowany funktor `./2` (kropka)[15], który odpowiada zdefiniowanemu wyżej `cons` oraz stała `[]` oznaczająca listę pustą. Co więcej, mamy możliwość zapisywania list w skrócie, tak jak w innych językach wysokiego poziomu, oraz dopasowywania głowy czy też ogona, a także dowolnej liczby elementów od przodu listy — o czym możemy przekonać się w trybie interaktywnym (Listing 6.7).

[15] W Prologu tradycyjnie *arność* (czyli liczbę argumentów) predykatu lub funktora oznacza się cyfrowo po jego nazwie, oddzieloną ukośnikiem — u nas więc byłoby wcześniej: `nil/0`, `cons/2`, `czyLista/1`, `jestGlowa/2`... Natomiast `./2` oznacza funktor lub predykat dwuargumentowy o nazwie złożonej ze znaku kropki.

Listing 6.7. Test wbudowanych list

```
?- X = .(1, .(2, .(3, [] ))).
X = [1, 2, 3].

?- [1, 2, 3, 4, 5] = [Glowa|Ogon].
Glowa = 1,
Ogon = [2, 3, 4, 5].

?- [1, 2, 3, 4, 5] = [G1, G2, G3|Ogon].
G1 = 1,
G2 = 2,
G3 = 3,
Ogon = [4, 5].

?- [1, [2, 3, 4], 5] = [G1, G2, G3|Ogon].
G1 = 1,
G2 = [2, 3, 4],
G3 = 5,
Ogon = [].
```

Spróbujmy więc zdefiniować predykat `permutacja/2`, który będzie prawdziwy, gdy druga lista będzie permutacją pierwszej. Definicje pokazuje Listing 6.8, a sprawdzić działanie Czytelnik zechce we własnym zakresie.

Listing 6.8. Permutacje list

```
permutacja([], []).
permutacja([G | O], P)
    :- permutacja(O, O1),
        wstawione(G, O1, P).

wstawione(X, L, [X | L]).
wstawione(X, [G | O], [G | O1])
    :- wstawione(X, O, O1).
```

Zdefiniujmy jeszcze — podobnie jak listy — binarne drzewa poszukiwań wraz z podstawowymi operacjami — Listing 6.9 (także do samodzielnego sprawdzenia).

Listing 6.9. Drzewo poszukiwań w Prologu

```
jestDrzewemBin(dPuste).
jestDrzewemBin(dBin(_, L, P))
    :- jestDrzewemBin(L),
        jestDrzewemBin(P).

elementDrzewaBin(X, dBin(X, _, _)).
elementDrzewaBin(X, dBin(_, L, _))
    :- elementDrzewaBin(X, L).
```

```
elementDrzewaBin(X, dBin(_, _, P))
    :- elementDrzewaBin(X, P).

wstawDoDrzewaBST(X, dPuste, dBin(X, dPuste, dPuste)).
wstawDoDrzewaBST(X, dBin(Y, L, P), dBin(Y, NL, P))
    :- X < Y,
       wstawDoDrzewaBST(X, L, NL).
wstawDoDrzewaBST(X, dBin(Y, L, P), dBin(Y, L, NP))
    :- X >= Y,
       wstawDoDrzewaBST(X, P, NP).

listaWDrzewoBST([], D, D).
listaWDrzewoBST([G | O], Przed, Po)
    :- wstawDoDrzewaBST(G, Przed, Posrednie),
       listaWDrzewoBST(O, Posrednie, Po).

noweDrzewoBSTZListy(Lista, D)
    :- listaWDrzewoBST(Lista, dPuste, D).

drzewoBSTWListe(dPuste, []).
drzewoBSTWListe(dBin(X, L, P), Lista) :-
    drzewoBSTWListe(L, ListaL),
    drzewoBSTWListe(P, ListaP),
    append(ListaL, [X | ListaP], Lista).
```

6.5. Arytmetyka i zagadki

Wcześniej (strona 125 w Podrozdziale 6.3.2) wspominaliśmy o arytmetyce w kontekście mało konstruktywnym. Jednak Prolog potrafi wykonywać obliczenia, choć nie za pomocą predykatu `=`, lecz predykatów infiksowych `=:=` (który wymaga wcześniej uzgodnionych obu argumentów i zwraca prawdę, jeśli wartości obu wyrażeń są równe co do wartości liczbowej) oraz `is` (który powinien mieć jako pierwszy argument zmienną nieuzgodnioną, a jako drugi — wyrażenie arytmetyczne; powoduje uzgodnienie danej zmiennej z wartością obliczoną z danego wyrażenia[16]).

Najpierw klasyka — definicja silni w Prologu (Listing 6.10). Zauważmy tutaj, że nie możemy użyć tej samej zmiennej do oznaczenia kilku wartości, bo wartość raz ustalona już się nie zmienia w tym samym wnioskowaniu. Stąd też w linii 3 mamy `Nminus1 is N - 1`, nigdy `N is N - 1` — to dru-

[16] A więc predykat `is` przypomina w pewnym sensie podstawienie z języków imperatywnych. Nie należy jednak dać się zwieść, bo wartości zmiennej raz uzgodnionej nie można już zmienić w obrębie klauzuli. Tak więc na przykład zapytanie `X is 1, X is X+1.` da w wyniku fałsz.

gie jest po prostu zawsze fałszywe! Podobnie, potrzebna jest nam zmienna pomocnicza `Spoprz`, nie możemy użyć tutaj `S`.

Listing 6.10. Silnia w Prologu

```
silnia(N, 1) :- N < 2.
silnia(N, S) :- N > 1,
                Nminus1 is N - 1,
                silnia(Nminus1, Spoprz),
                S is Spoprz * N.
```

Spróbujmy teraz rozważyć następującą zagadkę matematyczną: w zapisie $NINE \times THREE = NEUF \times TROIS$ poszczególne słowa oznaczają liczby w zapisie dziesiętnym, a poszczególne litery to cyfry od 0 do 9 tak, by dokładnie jednej cyfrze odpowiadała dokładnie jedna litera[17].

Jej rozwiązanie w Prologu jest banalne — Listing 6.11 (korzystamy tu z predykatu `permutacja` zdefiniowanego w Listingu 6.8).

Listing 6.11. Praktyczne zastosowanie arytmetyki w prologu

```
kryptarytm(E, F, H, I, N, O, R, S, T, U) :-
    permutacja([1,2,3,4,5,6,7,8,9,0],
               [E, F, H, I, N, O, R, S, T, U]),
    (N*1000+I*100+N*10+E*1)
          * (T*10000+H*1000+R*100+E*10+E*1)
       =:= (N*1000+E*100+U*10+F*1)
          * (T*10000+R*1000+O*100+I*10+S).
```

Szukanie wyniku (Listing 6.12) chwilę trwa (bo metoda jest oparta na brutalnej sile), ale zwartość powyższego zapisu jest tego warta:

Listing 6.12. Rozwiązanie kryptarytmu

```
?- kryptarytm(E, F, H, I, N, O, R, S, T, U).
E = 6,
F = 8,
H = 3,
I = 0,
N = 9,
O = 5,
R = 2,
S = 7,
T = 1,
U = 4 .
```

[17] Taka zagadka nazywa się *kryptarytmem*. Oczywiście, równanie w zagadce odczytane dosłownie także jest poprawne (obie strony to 9×3, lewa po angielsku, prawa po francusku).

I na koniec jeszcze łamigłówka logiczna (za [10]), którą z łatwością rozwiązuje Prolog:

> Ścigało się pięciu przyjaciół. Wincenty nie wygrał. Grzegorz był trzeci, poszło mu słabiej niż Dymitrowi, który nie był drugi. Andrzej nie był pierwszy ani ostatni. Borys był tuż za Wincentym. Kto był który na mecie?

Zapisujemy po prostu podane stwierdzenia w Prologu (Listing 6.13 — tu także wykorzystujemy `permutację` z Listingu 6.8) i samo wychodzi (Listing 6.14).

Listing 6.13. Algorytm rozwiązujący zagadkę

```
wyscigi(W, G, D, B, A) :-
    permutacja([W, G, D, B, A], [1, 2, 3, 4, 5]),
    W \= 1,
    G = 3,
    D < G,
    D \= 2,
    A \= 1,
    A \= 5,
    B =:= W+1.
```

Listing 6.14. Rozwiązanie zagadki

```
?- wyscigi(W, G, D, B, A).
W = 4,
G = 3,
D = 1,
B = 5,
A = 2 .
```

6.6. Pytania i zadania

1. Wyjaśnij pojęcia: klauzula, klauzula Horna, SLD-rezolucja, SLD-drzewo, WAM, fakt, zależność, reguła, cel, predykat, funktor, unifikacja, uzgodnienie.
2. Jaka jest różnica między predykatami `=`, `=:=` oraz `is`?
3. Co możesz powiedzieć o `X` w poniższych celach i o prawdziwości tych celów?

```
?- X = [X].
?- X = [1, 2, 3 | X].
?- X = 2*3.
```

```
?- X is 2*3.
?- X =:= 2*3.
?- X is X+1.
?- X is 1, X is X+1.
?- X = X+1.
?- X = 1, X = X+1.
```

4. Narysuj SLD-drzewo dla celu:

```
?- jesteElementem(X [1, 2, 3, 4]).
```

przy danych definicjach:

```
jesteElementem(G, [G | _]).
jesteElementem(H, [_ | R]) :- jesteElementem(H, R).
```

5. Rozważmy cel:

```
?- f(A, B) = f(B, f(C, D)).
```

oraz trzy klauzule:

```
A = B.
A = f(C, D), B = f(C, D).
A = f(1, 2), B = f(1, 2).
```

Która z nich stanowi najlepsze uzgodnienie dla powyższego celu i dlaczego?

6. Zdefiniuj predykat `odwrocone/2` prawdziwy wtedy i tylko wtedy, gdy oba argumenty są listami i jedna z nich ma te same elementy co druga, ale w odwrotnej kolejności.
7. * Wróćmy do silni (Listing 6.10 na stronie 129). Prolog — podobnie jak języki funkcyjne — umie optymalizować rekurencję ogonową (patrz strona 91, Podrozdział 5.5). Spróbuj więc samemu zdefiniować silnię w Prologu w wersji ogonowej. Wzoruj się na Listingu 5.2 ze strony 89.
8. * Zajrzyj do [10] i spróbuj rozwiązać przy pomocy Prologa kilka zagadek.

Rozdział 7

Język Python

7.1. Cechy szczególne

Niemalże na końcu skryptu chcemy przedstawić krótki przegląd języka *Python*[1]. Nadaje się on bowiem jako wyśmienity przykład wielu paradygmatów programowania, a właściwie ich gładkiego połączenia. Jakie cechy języka zadecydowały o poświęceniu mu tutaj osobnego rozdziału?

Język obiektowy. Python jest językiem całkowicie obiektowym — to znaczy, że wszystko (także zwykłe liczby, ale i typy[2], funkcje oraz moduły!) jest obiektem i każdy obiekt ma jednolity interfejs w postaci metod i własności (oczywiście różnych). W tym sensie Python — podobnie jak Smalltalk — jest bardziej obiektowy niż Java, a tym bardziej niż C++.

Programowanie strukturalne. Co ciekawe, mimo powyższego, Python może być używany jako język strukturalny (choć w niektórych sytuacjach bez wywołania metod raczej trudno się obyć).

Podejście funkcyjne. W końcu Python zawiera wiele narzędzi, które umożliwiają stosowanie paradygmatu funkcyjnego z jego istotnymi elementami (jak choćby funkcje działające na funkcjach, domknięcia, struktury nieskończone).

7.2. Cechy wyróżniające

Inne ważne cechy Pythona krótko przedstawiamy poniżej. Cechy te czynią z Pythona także niezły język do nauki programowania, i to w różnych paradygmatach.

Tryb interaktywny. Oprócz zwyczajnie uruchamianych programów w Pythonie[3] mamy także możliwość trybu interaktywnego, w który wchodzimy uruchamiając Pythona bez podanego żadnego pliku do wykonania. W takim trybie pracy wszystkie instrukcje działają normalnie, a te, które wymagają podania kolejnych wierszy (jak instrukcje złożone z wcięciami — patrz strona 136 — lub kontynuacje linii) są przez interpreter rozpoznawane

[1] Przykłady prezentowane tutaj dotyczą Pythona w wersji 2.5.2, ale w nowszych wersjach gałęzi 2 także powinny działać bez problemów. Jednak Python 3 nie jest kompatybilny wstecz, więc przykłady mogą wymagać pewnego dostosowania [57]. Warto też przy okazji nadmienić, że ten rozdział z założenia nie jest kursem tego języka, lecz przeglądem jego ciekawszych cech! Do bardziej systematycznej i pełnej nauki możemy polecić [12, 29, 33, 43, 50, 69, 81]...

[2] Przy okazji: typy, inaczej niż w Haskellu, są tu także wartościami pierwszego rzędu, a więc nie ma w Pythonie osobnego metasystemu typów. Ma to swoje plusy i minusy.

[3] Program w Pythonie jest zwykłym plikiem tekstowym (tradycyjnie z rozszerzeniem `py`). Żeby go wykonać, uruchamiamy go przez `python plik.py` — program jest kompilowany (a potem interpretowany) „w locie", nie ma potrzeby wcześniejszej kompilacji.

i użytkownik ma możliwość wpisania dalszych linii tekstu, a pusta linia jest sygnałem zakończenia takiego wielolinijkowego bloku. Ponadto w trybie interaktywnym instrukcje złożone z samego wyrażenia powodują wyświetlenie reprezentacji wyniku (o ile wynik nie jest pusty — `None`, patrz strona 142, Podrozdział 7.3.1.1) — normalnie wynik takiego wyrażenia jest (jak w C) ignorowany.

Przykładową sesję w trybie interaktywnym pokazuje Listing 7.1. Znaki `>>>` oraz `...` są znakami zachęty, po których użytkownik wpisuje wyrażenia i instrukcje (lub ich kontynuacje).

Listing 7.1. Sesja interaktywna z Pythonem

```
Python 2.5.2 (r252:60911, Jan 24 2010, 14:53:14)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> 3+4
7
>>> 12/7
1
>>> 12.0/7.0
1.7142857142857142
>>> 'Ala' + "␣ma␣" + '''kota.'''
'Ala␣ma␣kota.'
>>> '␣'.join(['Ala', 'ma', 'kota.'])
'Ala␣ma␣kota.'
>>> print '␣'.join(['Ala', 'ma', 'kota.'])
Ala ma kota.
>>> for i in range(4):
...     print i, i**2, i**(0.5)
...
0 0 0.0
1 1 1.0
2 4 1.41421356237
3 9 1.73205080757
>>> exit()
```

Prostota i zwartość. Choć to może się wydawać kwestią gustu, programy w Pythonie są jednak zwykle krótkie, bez zbędnych znaczków i konstrukcji mających znaczenie jedynie składniowe; przykładem mogą być Listingi 7.2 oraz 7.3. Warto ostatni z nich porównać z tym samym algorytmem zapisanym w Javie — Listing 7.4 (Listingi 7.3 oraz 7.4 na podstawie [13]).

Listing 7.2. *Hello, World!* w Pythonie

```
print "Hello, World!"
```

Listing 7.3. Przykładowy program w Pythonie (por. Listing 7.4)

```
aList = []
for i in range(1, 10):
  aList += [i]
aNumber = aList[3]
```

Listing 7.4. Przykładowy program w Javie (por. Listing 7.3)

```
public class Programik {
  public static void main (String[] args) {
    public Vector<Integer> aList = new Vector<Integer>;
    public int aNumber;
    for (int i = 1; i < 10; i++) {
      aList.addElement(i);
    }
    aNumber = aList.getElement(3);
  }
}
```

Określone kodowanie znaków. Pliki źródłowe muszą zawierać podaną wprost informację o zastosowanym kodowaniu znaków w postaci komentarza[4] (w pierwszej lub drugiej linii pliku), takiego jak:

```
# coding=utf-8
```

W przeciwnym razie przyjmowane jest kodowanie domyślne ASCII[5]. My w przykładach ograniczymy się jedynie do domyślnego zestawu znaków (by nie wydłużać przykładowych kodów), stąd brak w listingach polskich liter.

Wcięcia. Składnia Pythona wymaga wcięć, które w innych językach nie mają znaczenia formalnego, ale oczywiście zwykle programiści uczą się je stosować dla zachowania czytelności programu. W Pythonie nie ma symboli ani słów kluczowych wydzielających blok instrukcji (jak `{...}` w C lub w Javie, czy też `begin...end` w Pascalu lub w Adzie), lecz o podziale tekstu programu na bloki decydują wcięcia. Proste zasady rządzące wcięciami są następujące:

— linie puste lub zawierające tylko komentarze i/lub znaki białe są ignorowane;
— pierwsza niepusta linia kodu nie może być wcięta;

[4] Komentarze w Pythonie rozpoczynają się znakiem `#` i rozciągają się do końca wiersza.

[5] A więc nie można używać znaków o kodach powyżej 127!

— jeśli jakaś linia kończy się dwukropkiem : (który w Pythonie rozpoczyna blok instrukcji), to następna linia **musi** być wcięta głębiej (co najmniej o jedną spację) niż ta z dwukropkiem;
— jeśli jakaś linia nie kończy się dwukropkiem, to następna linia **nie może** być głębiej wcięta — musi być wcięta tak samo jak ta bez dwukropka (i oznacza to kontynuację bloku), albo też wcięta płycej, jednakże nie dowolnie, lecz dokładnie tak, jak któryś z dotychczas niezamkniętych bloków (i oznacza to zakończenie wszystkich bloków wciętych głębiej). Przykład poprawnego stosowania tych zasad dostarcza Listing 7.5.

Listing 7.5. Poprawne wcięcia

```
a = float(raw_input("Podaj pierwsza liczbe: "))
b = float(raw_input("Podaj druga liczbe: "))
if a > b:
    max = a
    min = b
else:
        max = b
        min = a
print ("Wieksza z tych liczb to %s, a mniejsza to %s."
            % (max, min))
```

Nie jest to przykład elegancki, bo wcięcia są różnych rozmiarów w różnych blokach[6], ale jest poprawny. Widać, że dwukropki w liniach 3 i 6 rozpoczynają nowy blok instrukcji i stąd wymuszają wcięcia w liniach 4 i 7; zaś linie 6 i 9 stanowią powrót do bloku nadrzędnego (tu: całego programu), ponieważ wracają do rozmiaru wcięcia, jaki był wcześniej.

Tu warto zwrócić też uwagę na linię 10, która jest kontynuacją linii 9. Skąd Python o tym wie? Nawias otwarty w linii 9 nie został zamknięty, więc następna linia jest ciągiem dalszym poprzedniej, a wcięcia w linii kontynuującej są ignorowane.

Listing 7.6 przedstawia ten sam program z niepoprawnymi wcięciami.

Listing 7.6. Niepoprawne wcięcia

```
    a = float(raw_input("Podaj pierwsza liczbe: "))
  b = float(raw_input("Podaj druga liczbe: "))
if a > b:
      max = a
    min = b
```

[6] Warto stosować konsekwentnie stały krok wcięcia — zwykle są to dwie, cztery lub osiem spacji. Niedobrze jest też stosować tabulatory, a jeśli już, to nie wolno ich mieszać ze spacjami we wcięciach, bo Python je rozróżnia i nie zawsze interpretuje tak, jak byśmy się tego spodziewali po wyglądzie programu w edytorze. Tak więc zawsze tu stosujemy spacje.

```
6   else:
        max = b
8       min = a
  print ("Wieksza z tych liczb to %s, a mniejsza to %s."
10 % (max, min))
```

Błędy są tu następujące:
— linia 1 jest wcięta niepotrzebnie;
— linie 2, 3, 5 są wcięte płycej od swoich poprzedniczek, ale nie może to być zakończeniem żadnego bloku, bo ich wcięcia nie pasują do żadnego otwartego bloku;
— linia 6 powinna być wcięta równo z linią 3 (`else` równo ze swoim `if`);
— linia 8 jest wcięta głębiej niż 7, ale w linii 7 nie ma (i nie powinno być) dwukropka.

Natomiast linia 10 jest w porządku, bo jest kontynuacją linii 9, więc w niej wcięcia mogą być dowolne.

Warto przy tej okazji zwrócić uwagę, że nie istnieje w Pythonie problem „wiszącego `else`" [2, 66]. Polega on na tym, że w językach, takich jak C, Pascal, Java, kiedy po dwóch frazach `if` mamy jedną frazę `else`, nie można zgodnie z definicją składniową języka określić jednoznacznie, do której frazy `if` należy owa fraza `else`. Zwykle przyjmuje się dodatkową regułę, że w takim przypadku należy ona do bliższego `if`. W Pythonie nie ma takich wątpliwości, o czym możemy się przekonać porównując Listingi 7.7 i 7.8 (w szczególności wiersze 7 w obu Listingach).

Listing 7.7. Niewiszące `else` I

```
  a = float(raw_input("Podaj pierwsza liczbe: "))
2 b = float(raw_input("Podaj druga liczbe: "))
  c = float(raw_input("Podaj trzecia liczbe: "))
4 if a > b:
    if a > c:
6     print "Najwieksza jest %s." % (a,)
    else:
8     print "%s > %s oraz %s <= %s." % (a, b, a, c)
```

Listing 7.8. Niewiszące `else` II

```
  a = float(raw_input("Podaj pierwsza liczbe: "))
2 b = float(raw_input("Podaj druga liczbe: "))
  c = float(raw_input("Podaj trzecia liczbe: "))
4 if a > b:
    if a > c:
6     print "Najwieksza jest %s." % (a,)
  else:
8   print "%s <= %s, ale co z %s?" % (a, b, c)
```

Instrukcja pusta. Ponieważ puste wiersze są nieznaczące, a jest czasem potrzeba zapisania instrukcji pustej (nic nierobiąca funkcja, pusta klasa, pusty refren pętli), Python dysponuje instrukcją pustą pisaną wprost: `pass`.

Typowe konstrukcje programistyczne. Listing 7.9 przedstawia wygląd podstawowych elementów języka — kaskadę selekcji, definicję funkcji, pętlę „dopóki", pętlę „ dla".

Listing 7.9. Podstawowe konstrukcje programistyczne

```
if warunek:
    ...
elif warunek:
    ...
elif warunek:
    ...
else:
    ...

def nazwa_funkcji(parametry):
    """Opcjonalny komentarz dokumentujacy.
    Moze miec wiele linii, w tym puste."""
    ...
    return wynik

while warunek:
    ...

for zmienna in sekwencja:
    ...

for zmienna in range(...):
    ...
```

W definicji funkcji może pojawić się (podobnie także w definicji klasy) *komentarz dokumentujący* umieszczany bezpośrednio po nagłówku definicji. Komentarz ów — by pełnił swoje specjalne funkcje — nie może być zapisywany jak zwykły komentarz po znaku #, lecz jako napis (zwykle wielowierszowy). Różni się od zwykłego komentarza tym, że nie jest zupełnie ignorowany, lecz zapamiętywany wraz z definicją i może być wykorzystany

do automatycznej generacji dokumentacji, jako opis danej funkcji (klasy), na przykład przez standardową funkcję `help`[7].

Jeszcze jednego objaśnienia wymaga tu postać instrukcji `for`, odmienna od spotykanych w innych językach pętli „dla", ale przypominająca występujące w niektórych językach konstrukcje `foreach`. W Pythonie zmienna sterująca pętli `for` przyjmuje kolejne wartości z podanej sekwencji (być może nieskończonej!). Podobną do tradycyjnej pętlę „dla" uzyskamy, generując sekwencję za pomocą funkcji `range()` (lub `xrange()`). Taka postać pętli jest jednak w Pythonie używana dużo rzadziej niż w innych językach — jest zwykle niepotrzebna, gdyż mamy do dyspozycji jej ogólniejszą postać.

Zbieranie nieużytków. Python — jak wiele nowoczesnych języków wysokiego poziomu — sam zarządza pamięcią i zwalnia pamięć nieużywanych obiektów przez odśmiecanie (patrz strona 56, Podrozdział 3.3.2.5). Nie ma więc konieczności świadomej kontroli alokacji i dealokacji pamięci ze strony programisty — wszystkie dane są alokowane niejawnie automatycznie na stercie, a pamięć zwalniana, gdy jest to możliwe i potrzebne.

Biblioteki. Python dysponuje masą wbudowanych modułów oraz pozwala na ich łatwe, samodzielne budowanie. Jedną z instrukcji umożliwiających korzystanie z zewnętrznego modułu jest
`import nazwa`
importująca moduł o podanej nazwie jako obiekt (moduły także są obiektami!), z którego zawartości można korzystać później przez odwołania kwalifikowane nazwą modułu (`nazwa.funkcja()`, `nazwa.zmienna` itp.).

Podejście do wyjątków. Python zaleca stosowanie zasady „łatwiej prosić o wybaczenie niż o pozwolenie" (EAFP)[8] zamiast preferowanej w innych językach przeciwnej zasady „patrz gdzie skaczesz" (LBYL, ang. *Look Before You Leap*. Polega ona na tym, że program piszemy tak, jakby miał się zawsze powieść, nie dbając o sprawdzanie poprawności danych i powodzenia operacji, natomiast błędy i inne problemy traktujemy jako sytuacje wyjątkowe i obsługujemy za pomocą... wyjątków. Wyjątki w Pythonie są reprezentowane przez całą, dość szczegółową hierarchię klas, które mogą być dowolnie

[7] Funkcja `help` jest bardzo przydatną funkcją standardową, korzystającą z refleksyjności (Podrozdział 7.5.7) Pythona: dla obiektu podanego jako argument wypisuje krótką pomoc, bazując przy tym na wspomnianym wyżej komentarzu dokumentującym oraz na samym nagłówku funkcji/klasy oraz na metodach klasy.

[8] Angielskie powiedzenie *it's Easier to Ask Forgiveness than it is to get Permission* przypisywane jest kontradmirał Grace Hopper [74], programistce i współtwórczyni jednego z pierwszych kompilatorów. Jej także zawdzięczamy popularyzację terminu *debugging* (odpluskwianie) w zastosowaniu do usuwania błędów z programów komputerowych (podobno znalazła mola w maszynie cyfrowej, na której pracowała).

rozszerzane. Ponadto mogą być przekazywane przez wiele kolejnych poziomów wywołań i struktur programu (nie tak, jak w starszych wersjach Javy), a więc dowolnie odkładane na później, bez przechwytywania i ponownego zgłaszania.

Porównanie wspomnianych dwóch podejść pokazują Listingi 7.10[9] oraz 7.11 — warto zwrócić uwagę na czystość, czytelność i zwartość[10] rozwiązania według zasady EAFP. Zasada EAFP ma w tym przypadku jeszcze jedną poważną zaletę nad LBYL: w Listingu 7.10 dwukrotnie wykonują się pewne czynności, bo dublujemy „ręcznie" funkcjonalność funkcji `int` (która i tak sprawdza poprawność formy zadanego napisu).

Listing 7.10. Zasada LBYL w praktyce

```
napis = raw_input("Podaj liczbe calkowita: ")
# pozbywamy sie spacji, ktore nie sa klopotem
# dla funkcji int(), ale sa dla naszego sprawdzania
napis = napis.strip()
# teraz sprawdzamy, czy podano liczbe
if len(napis) > 0 and napis[0] in ['-', '+']:
  znak = int(napis[0] + '1')
  napis = napis[1:].strip()
else:
  znak = 1
jest_ok = len(napis) > 0
for znak in napis:
  if znak not in ['0', '1', '2', '3', '4',
                  '5', '6', '7', '8', '9']:
    jest_ok = False
    break
# w zaleznosci od wyniku testu wykonujemu rozne czynnosci
if jest_ok:
  print "Szescian podanej liczby to %s." % (
          (znak*int(napis))**3,)
else:
  print "Bledny zapis liczby!"
```

Listing 7.11. Zasada EAFP w praktyce

```
napis = raw_input("Podaj liczbe calkowita: ")
# nic nie sprawdzamy recznie, tylko probujemy
try:
```

[9] Oczywiście Listing 7.10 jest dlatego tak bardzo rozbudowany, bo chcieliśmy osiągnąć możliwie dokładnie funkcjonalność Listingu 7.11.

[10] A jak się bardziej zastanowić to i uniwersalność, bo zamiana napisu na liczbę rzeczywistą (zamiast na liczbę całkowitą) w Listingu 7.11 wymaga tylko zamiany funkcji `int` na `float`; zmiany potrzebne w Listingu 7.10 są dużo poważniejsze i, być może, wielokrotnie wydłużające ów kod.

```
4     print "Szescian podanej liczby to %s." % (int(napis)**3,)
  except ValueError:
6     print "Bledny zapis liczby!"
```

7.3. Typy i zmienne w Pythonie

Python jest językiem *bardzo silnie typowanym* (patrz strona 45, Podrozdział 3.3.1). Jednakże zmienne nie są deklarowane w Pythonie, a właściwie deklaracja zmiennych odbywa się w momencie podstawienia. W przeciwieństwie jednak do wielu języków silnie typowanych, Python jest językiem z dynamicznymi wiązaniami (patrz Podrozdział 3.1), w szczególności z dynamicznym wiązaniem zmiennej z jej typem. Oznacza to, że każde podstawienie może zmienić typ zmiennej — Listing 7.12 jest poprawny.

Listing 7.12. Dynamiczne wiązanie typów w Pythonie (sesja interaktywna)

```
  >>> x = 12
2 >>> print x+1
  13
4 >>> x = 'ala'
  >>> print x+'rm'
6 alarm
```

Jednakże (pomimo wspomnianego wyżej bardzo silnego typowania), bardzo rzadko dochodzi do sprawdzania typów wprost (i, choć możliwe, nie jest to zalecane także programującym w Pythonie[11]). Stosowana jest za to zasada zgodności sygnatur (czy też interfejsów metod) w dostępie do obiektów — o czym więcej w Podrozdziale 7.5.

7.3.1. Wbudowane typy

Python dysponuje szerokim asortymentem typów wbudowanych. Nie wszystkie z nich tutaj wymienimy, ale jedynie te, które mogą przydać się w niniejszym rozdziale.

7.3.1.1. Typy podstawowe

[11] Choć i w naszych przykładach się przyda — wiersz 3 Listingu 7.37.

Typ jednostkowy. W Pythonie nazywa się on `NoneType`[12] i jego jedyną wartością jest `None`. Wartość ta oznacza oczywiście... brak wartości. Odpowiada typowi jednostkowemu w Haskellu (strona 107 w Podrozdziale 5.8.1).

Inne typy podstawowe. Możemy do nich zaliczyć między innymi `object` (nadtyp wszystkich typów), `FunctionType` (typ wszystkich funkcji) oraz `ClassType` i `type`. Dwa ostatnie to *metaklasy*, czyli typy będące zbiorami typów (klasy klas) — pierwsza to zbiór klas starego stylu, druga to zbiór klas nowego stylu (patrz strona 160, Podrozdział 7.5.1).

7.3.1.2. Liczby

„Zwykłe" liczby. Mamy w Pythonie liczby zwyczajne (to jest znane powszechnie z innych języków programowania), a więc normalne liczby całkowite `int`, na przykład `34`, a także tradycyjne liczby zmiennoprzecinkowe `float`, na przykład `4.12`. Do liczb zaliczyć należy także typ logiczny `bool` o dwóch możliwych wartościach `True` oraz `False`, którym odpowiadają w wyrażeniach arytmetycznych liczby `1` i `0`.

Długie liczby całkowite. Python może jednak operować też na liczbach całkowitych dowolnej długości (typ `long`), ograniczonej jedynie dostępną pamięcią. Tak więc wynikiem instrukcji `print 2**100` (podwójna gwiazdka oznacza potęgowanie) jest wyświetlenie wartości
`1267650600228229401496703205376L`
(końcowe `L` oznacza właśnie literał typu `long`).

Liczby zespolone. W końcu Python ma wbudowane także liczby zespolone (`complex`), tak więc wykonanie `print (3+2j) / (1+1j)` wyświetli `(2.5-0.5j)`.

7.3.1.3. Sekwencje

Jak za chwilę zobaczymy, semantyka list i krotek różni się znacznie od typów o tych samych nazwach znanych nam już z Haskella (Podrozdział 5.6).

Listy. Python nie ma wbudowanego tradycyjnego typu tablicowego, natomiast odpowiednikiem tablic (dynamicznych) są listy (`list`) mogące mieć elementy różnych typów. Różnią się one od tradycyjnego podejścia do list

[12] Typ każdej danej `x` można zawsze uzyskać — jako obiekt — wywołaniem funkcji `type(x)`, jednak niektóre typy są od razu dostępne pod swoimi nazwami klas (z tych, które tu wymieniamy są to te pisane małymi literami), a pozostałe wymagają importu modułu `types`.

znanego z języków funkcyjnych (Podrozdział 3.3.2.6 na stronie 60, Podrozdział 5.6.1 na stronie 92) i programowania logicznego (Podrozdział 6.4 na stronie 125), gdzie lista niepusta jest definiowana rekurencyjnie jako para składająca się z elementu i kolejnej listy — a więc jest struktura o dostępie sekwencyjnym, a nie swobodnym, jak tablica. Tutaj lista jest zdefiniowana wewnętrznie, a implementacja jest częściowo tablicowa, i wiele operacji znanych z tablic (jak zmiana lub odczyt któregoś z elementów), ale i innych przydatnych (jak usunięcie/dodanie elementu lub kilku, połączenie list) jest dzięki temu możliwych do wykonania w rozsądnym czasie. Przykładowe listy i operacje na nich (w tym wycinki list) pokazuje Listing 7.13.

Krotki. Krotki (`tuple`) są niezmienialne (patrz Podrozdział 7.3.3) — można je traktować jako listy[13], których zmienić nie można, używamy więc ich tylko do odczytu. Są za to nieco szybsze i można używać ich tam, gdzie wymagane są dane niezmienialne.

Listing 7.13. Listy i krotki w Pythonie

```
# listy
lpusta = []
l1 = [1]
l1.append(2)
l2 = [l1, lpusta, "napis", ["lista", "napisow"]]
l3 = l1+l2+l1
print 100*["x"], l1, l2, l3, lpusta, l3[1]
l4 = [0, 1, 2, 3, 4, 5, 6, 7]
print l4[3:5], l4[4:], l4[:3], l4[3:-1]
print l4[1:6:2], l4[::2], l4[6:1:-2], l4[::-1]
# krotki
kpusta = ()
k1 = (4,)    # to jest poprawna krotka jednoelementowa
             # (przecinek konieczny!)
x = (4)      # a to nie jest w ogole krotka!
k2 = 3*(kpusta,)
k3 = k1+k1+(lpusta, l1)
print kpusta, k1, k2, k3, x, k3[2]
# konwersja
print list(k2), tuple(l2)
```

Napisy. Napisy (`str`) są niezmienialnymi sekwencjami znaków. Można zapisywać je na różne sposoby. Najbardziej odbiegającym od tradycyjnego zapisu jest zapis wielowierszowy. Podobnie jak kontynuacja logicznej linii

[13] A więc znowu, podobnie jak lista, krotka także znaczy trochę co innego niż w Haskellu.

programu w kolejnych wierszach, gdy pozostaje niezamknięty nawias, możliwe jest rozciągnięcie napisu na wiele linii — o ile do otwarcia i zamknięcia napisu użyjemy potrójnych apostrofów lub potrójnych cudzysłowów. Napis tak skonstruowany zachowuje podział na wiersze.

Listing 7.14. Napis wielowierszowy

```
print """To
jest napis
    wielowierszowy..."""
```

Tak więc wykonanie programu z Listingu 7.14 wyświetli:

```
To
jest napis
    wielowierszowy...
```

Inna, godna wspomnienia, rzecz związana z napisami to operator formatowania `%` używany już w przykładach w tym rozdziale. Używamy go między innymi w postaci `napis % krotka`, gdzie `napis` jest napisem formatującym (zawierającym sekwencje formatujące, z których najprostsza i najogólniejsza to `%s`), natomiast `krotka` to krotka zawierająca dane wstawiane do napisu w miejsce sekwencji formatujących.

7.3.1.4. Słowniki

Jeszcze jedną zmienialną strukturą danych są słowniki (`dict`), które stanowią Pythonowy odpowiednik tablic asocjacyjnych (strona 61, Podrozdział 3.3.2.7). Mogą one w roli klucza mieć dowolne obiekty niezmienialne głęboko — czyli takie, które są niezmienialne wraz ze swoimi wszystkim elementami. Podstawowe operacje na słownikach prezentuje Listing 7.15

Listing 7.15. Słowniki i operacje na nich (tryb interaktywny)

```
>>> s1 = {}
>>> s1['ala'] = 15
>>> s1['jan'] = 12
>>> s2 = {'a': 10, 10: 'a'}
>>> s3 = {'napis': 5, ('kro', 'tka'): 2, None: 0}
>>> print s1
{'jan': 12, 'ala': 15}
>>> print s1.values()
[12, 15]
>>> del s1['ala']
>>> print s1
{'jan': 12}
>>> print s2
{'a': 10, 10: 'a'}
>>> print s2.items()
```

```
    [('a', 10), (10, 'a')]
17 >>> print s3
    {'napis': 5, None: 0, ('kro', 'tka'): 2}
19 >>> print s3.keys()
    ['napis', None, ('kro', 'tka')]
```

7.3.2. Zmienne, czyli nazwy obiektów

W Pythonie nie powinniśmy traktować zmiennych jako pudełek, w których przechowujemy dane — jak często jest to obrazowo opisywane przy okazji innych języków. Wszystkie obiekty w Pythonie identyfikowane są przez (niejawne) referencje, a zmienne przechowują tylko te referencje. Obiekty natomiast istnieją w pamięci całkiem niezależnie od zmiennych. Tak więc zmienne można traktować jako nazwy obiektów, przy czym mogą istnieć obiekty nienazwane, jak i obiekty o wielu nazwach.

Wraz z kolejną cechą, to jest opisaną poniżej (Podrozdział 7.3.3) zmienialnością niektórych danych, może to stwarzać sytuacje zaskakujące, szczególnie początkujących adeptów programowania w Pythonie. Dlatego też mamy pewne narzędzia, które pozwalają na rozróżnienie obiektów.

— Operator `is` odpowiada na pytanie o **tożsamość** dwóch obiektów. W odróżnieniu od porównania `==` — które przyrównuje wartości obiektów (a przynajmniej tak jest zaimplementowane w typach standardowych) — operator `is` porównuje jedynie referencje dwóch obiektów, a więc sprawdza równość ich adresów.
— Funkcja `id` zwraca liczbę całkowitą będącą identyfikatorem obiektu podanego jako argument. Identyfikator ten jest w danej chwili unikalny dla danego obiektu, jednakże w czasie pracy programu (lub tym bardziej w kolejnych wywołaniach) identyfikator obiektu może się zmienić, albo nowy obiekt może dostać identyfikator po istniejącym wcześniej obiekcie. Technicznie `id` jest po prostu adresem obiektu przedstawionym jako długa liczba całkowita.

Przykłady ustalania tożsamości obiektów pokazuje Listing 7.16[14], zaś Rysunek 7.1 ilustruje sytuację po wykonaniu tego kodu (prostokątami zaokrąglonymi oznaczyliśmy nazwy zmiennych, prostokątami ostrymi — obiekty, strzałki przedstawiają referencje). Jak widać, mamy tu do czynienia z wielokrotnym aliasowaniem.

[14] W linii 5 widać *porównanie łańcuchowe* (możliwe także z użyciem nierówności, także różnych, w jednym wyrażeniu) — skrótowy zapis koniunkcji porównań znany z matematyki, ale rzadko dopuszczalny w językach programowania. W Pythonie jest jak najbardziej poprawny.

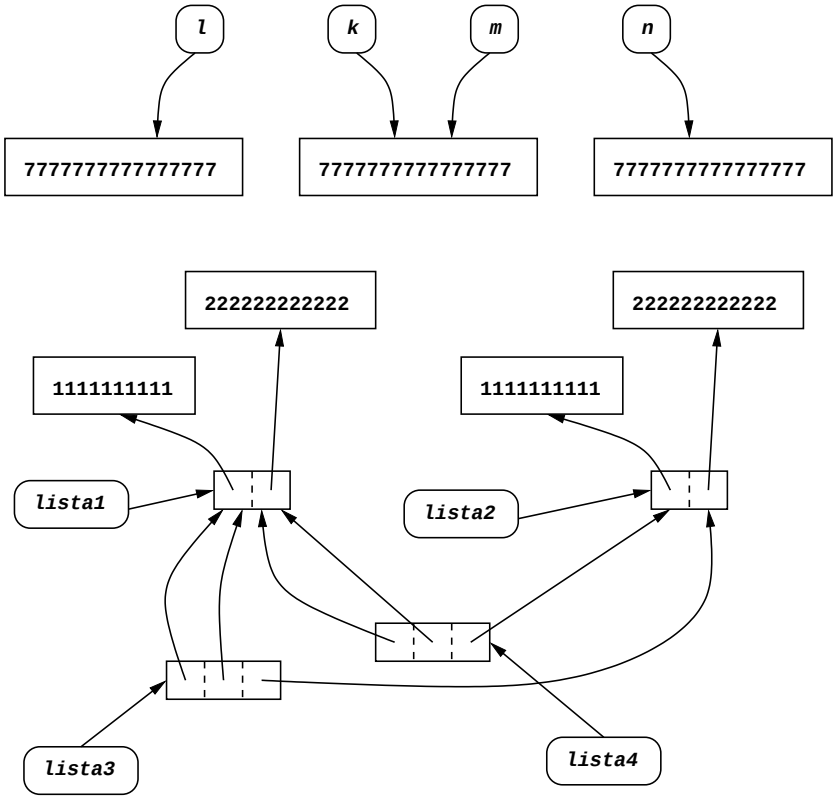
Listing 7.16. Użycie `is` do identyfikacji obiektów

```
>>> k = 7777777777777777
>>> l = 7777777777777777
>>> m = k
>>> n = 1+7777777777777776
>>> k == l == m == n
True
>>> k is l
False
>>> k is m
True
>>> k is n
False
>>> n is l
False
>>> lista1 = [1111111111, 222222222222]
>>> lista2 = [1111111111, 222222222222]
>>> lista1 is lista2
False
>>> lista1[0] is lista2[0]
False
>>> lista1[1] is lista2[1]
False
>>> lista3 = [lista1, lista1, lista2]
>>> lista4 = [lista1, lista1, lista2]
>>> lista3 is lista4
False
>>> lista3[0] is lista4[0]
True
>>> lista3[1] is lista4[1]
True
>>> lista3[0] is lista3[1]
True
```

W powyższym przykładzie specjalnie użyliśmy dużych liczb, bowiem małe liczby całkowite traktowane są specjalnie. Spróbujmy przyjrzeć się pierwszym liniom tego kodu z użyciem mniejszych liczb — Listing 7.17 oraz Rysunek 7.2. Widzimy tutaj, że małe liczby całkowite traktowane są nieco inaczej. Otóż, Python w momencie uruchomienia generuje pewne często używane wielkości i utrzymuje je przez cały czas działania w pewnym rodzaju słowniku — nie stosując do nich zasad zbierania nieużytków, ani nie generując ich wielokrotnie, lecz dowiązując te istniejące z owego słownika w miarę potrzeb. Tak dzieje się z — często przecież używanymi — małymi liczbami całkowitymi.

Listing 7.17. Tożsamość małych liczb całkowitych

```
>>> k = 7
```

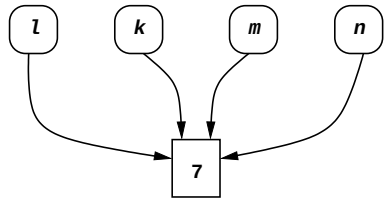


Rysunek 7.1. Zmienne, nazwy, obiekty z Listingu 7.16

```
>>> l = 7
>>> m = k
>>> n = 1+6
>>> k == l == m == n
True
>>> k is l
True
>>> k is m
True
>>> k is n
True
```

7.3.3. Zmienialność i niezmienialność danych

Powyższe uwagi byłyby tylko ciekawostką, gdyby nie to, że w Pythonie niektóre dane są *zmienialne*. Cóż to tak naprawdę znaczy?



Rysunek 7.2. Zmienne, nazwy, obiekty z Listingu 7.17

Otóż, wszystkie dane w Pythonie dzielą się na obiekty zmienialne[15] i obiekty niezmienialne. Na obiektach niezmienialnych możemy wykonywać tylko operacje typu czytania (jak na krotkach, napisach, liczbach), a gdy wydaje nam się, że zmienia się wartość takiego obiektu, to tak naprawdę tworzony jest całkiem nowy byt[16]. To zachowanie ilustruje przykład sesji interaktywnej przedstawiony na Listingu 7.18 oraz 7.19 (liczby, krotki, napisy są obiektami niezmienialnymi).

Listing 7.18. Próba zmiany obiektów niezmienialnych (liczb i napisów)

```
>>> a = 7
>>> b = a
>>> a += 10
>>> print a, b
17 7
>>> s = 'napis'
>>> t = s
>>> s += 'ik'
>>> print s, t
napisik napis
```

Listing 7.19. Próba zmiany obiektów niezmienialnych (krotek)

```
>>> k1 = (1, 2)
>>> k2 = k1
>>> k1 += (3, 4)
>>> print k1, k2
(1, 2, 3, 4) (1, 2)
```

[15] Brzmi to nieco niezręcznie (ładniej brzmiało by: obiekty zmienne), ale nie mówimy tu o zmiennych w informatycznie normalnym, rzeczownikowym sensie tego słowa (ang. *variable*), lecz w przymiotnikowym — o obiektach, które mogą być zmieniane (ang. *mutable*). Dlatego „zmienialne", żeby nie myliło się... Spotyka się też określenie „mutowalne", ale to już jakoś całkiem nieładnie...

[16] Obiekty niezmienialne odpowiadają trwałym danym znanym z języków czysto funkcyjnych (patrz strona 99 w Podrozdziale 5.6.2.1).

Inaczej jest z obiektami zmienialnymi. Tutaj instrukcje zmieniające obiekt działają w miejscu, to jest na danym obiekcie. Szczególnie widoczne jest to przy operatorach takich, jak `+=` i jemu podobne — patrz Listing 7.20.

Listing 7.20. Zmiany obiektów zmienialnych (list)

```
>>> l1 = [1, 2]
>>> l2 = l1
>>> l1 += [3, 4]
>>> print l1, l2
[1, 2, 3, 4] [1, 2, 3, 4]
```

Na Listingu 7.20 widzimy operacje na listach (listy są zmienialne) całkiem analogiczne do operacji na krotkach z Listingu 7.19, ale wynik jest zgoła inny! Dlaczego tak jest? W obu kodach w pierwszych dwóch wierszach sytuacja jest taka sama: mamy jeden obiekt (w 7.19 krotkę, w 7.20 listę) z dwiema nazwami. Jednakże po wykonaniu wiersza 3 w kodzie 7.20 ten obiekt zmienia się, bo może — lista jest zmienialna, a więc doczepiamy do niej kolejne elementy. Nie jest więc tworzony nowy obiekt, a wszystko dzieje się na tej pierwotnej liście. Inaczej w Listingu 7.19 — tam krotka (jako niezmienialna) nie może się powiększyć o nowe elementy, więc tworzona jest nowa, złożona z czterech elementów i jej referencja jest dowiązywana do zmiennej `k2`. Natomiast zmienna `k1` nie jest wcale zmieniana, więc dalej wskazuje na pierwotną — niezmienioną! — krotkę.

Zmienialność może wprowadzić też dodatkowe komplikacje w połączeniu z wszechobecnym, „głębokim” aliasowaniem. Rozważmy sytuację, w której mamy dany napis, o którym wiemy, że jest złożony z samych cyfr i mamy za zadanie przejrzeć go i zapamiętać do jakiejś dalszej obróbki pozycje poszczególnych cyfr. Wygodnym rozwiązaniem byłoby użycie w roli wyniku listy, która na poszczególnych pozycjach (odpowiadających wartościom odpowiednich cyfr) zawierałaby listy pozycji kolejnych cyfr w danym napisie. Na przykład, dla napisu `"1451"` wynikiem byłaby lista:
`[[], [0, 3], [], [], [1], [2], [], [], [], []]`.

Narzucające się rozwiązanie przedstawia Listing 7.21.

Listing 7.21. Szukanie pozycji cyfr — podejście pierwsze

```
napis = raw_input("Podaj napis zlozony z samych cyfr: ")
wynik = 10*[[]]
for (i, z) in enumerate(napis):
    wynik[int(z)] += [i]
print wynik
```

Krótkie wyjaśnienia:

— wiersz drugi tworzy listę dziesięciu list pustych;

— w wierszu trzecim zamiast pojedynczej zmiennej licznikowej pętli mamy krotkę (parę) zmiennych, bo funkcja `enumerate` zwraca sekwencję złożoną z par `(indeks, element)` dla każdego elementu danej sekwencji (tu: napisu); stąd zmienne `i` oraz `z` będą przybierały w kolejnych obrotach pętli odpowiednio: numer bieżącego znaku i jego wartość;
— w końcu wiersz czwarty dopisuje numer pozycji bieżącego znaku do listy w głównej liście wynikowej o indeksie równym wartości tego znaku.

Sprawdźmy, jak działa nasz program dla przykładowego napisu `"1451"`. Niestety, zamiast spodziewanego:

```
[[], [0, 3], [], [], [1], [2], [], [], [], []]
```

dostajemy:

```
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3],
 [0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3],
 [0, 1, 2, 3], [0, 1, 2, 3]]...!
```

Cóż się stało?

Problem leży w wierszu drugim. Sytuację po jego wykonaniu przedstawia Rysunek 7.3. Dzieje się tak dlatego, że lista pusta jest tu tworzona jednokrotnie, a jedynie jej referencja jest dziesięciokrotnie wstawiana do listy zewnętrznej — a więc wszystkie jej indeksy wskazują na tę samą listę! Stąd wiersz czwarty dodaje pozycję elementu ciągle do jednej i tej samej listy.

Potrzebujemy tak przerobić wiersz drugi, by mieć dziesięć różnych pustych list, czyli sytuację, jak na Rysunku 7.4. Trzeba więc wymusić utworzenie dziesięciu pustych list. Jeden ze sposobów poprawienia kodu przedstawia Listing 7.22. Czytelnik zechce sam przeanalizować i sprawdzić w praktyce działanie tego kodu.

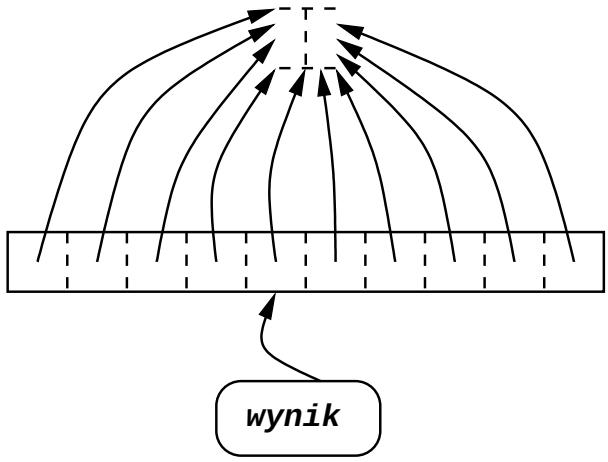
Listing 7.22. Szukanie pozycji cyfr — po poprawkach

```
1 napis = raw_input("Podaj napis zlozony z samych cyfr: ")
  wynik = []
3 for i in range(10):
    wynik += [[]]
5 for (i, z) in enumerate(napis):
    wynik[int(z)] += [i]
7 print wynik
```

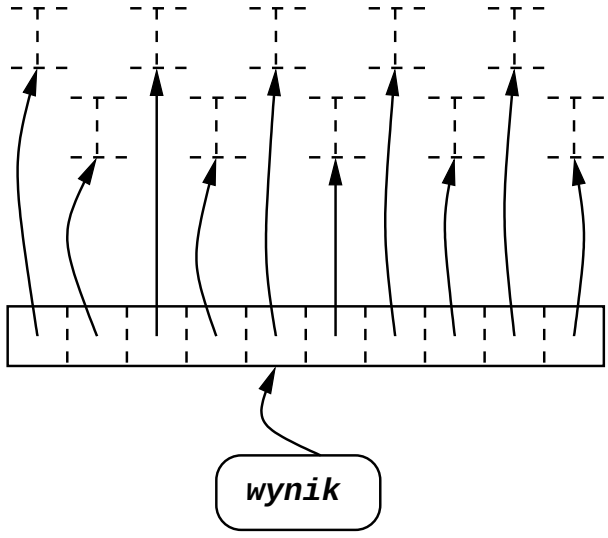
7.4. Podprogramy w Pythonie

Wszystkie podprogramy w Pythonie zwane są funkcjami[17] i zawsze zwracają jakiś wynik (a dokładniej, oczywiście referencję do pewnego obiektu;

[17] Są też generatory, ale o nich w Podrozdziale 7.6.



Rysunek 7.3. Sytuacja po drugim wierszu z Listingu 7.21 (przerywany symbol oznacza tu pustą listę)



Rysunek 7.4. Sytuacja wymagana przed trzecim wierszem z Listingu 7.21

warto zauważyć, że nie ma **absolutnie żadnego ograniczenia** na typ zwracanego obiektu). Do zwracania (i jednocześnie przerywania wykonywania funkcji) służy instrukcja `return`, której parametrem jest zwracana wartość. Parametr ten jest opcjonalny i jego wartością domyślną jest `None`. Przykład definicji i wywołań dwóch funkcji bezparametrowych przedstawia Listing 7.23 (linię 3 można opuścić bez zmiany znaczenia kodu; można w niej też opuścić samo słowo `None`).

Listing 7.23. Prosta funkcja w Pythonie

```
1 def witaj1():
    print "Hello, world!"
3   return None

5 def witaj2():
    return "Hello, world!"
7
  witaj1()
9 print witaj2()
```

7.4.1. Parametry formalne

Python należy do języków z bardzo różnorodnymi (i jednocześnie uniwersalnymi) sposobami przekazywania parametrów do procedur. Warto jednak zauważyć, że są one **zawsze parametrami wejściowymi** (patrz strona 70 w Podrozdziale 4.1.3.1). Jednakże — ponieważ (podobnie jak w przypadku zwykłych zmiennych) przechowują one referencje do obiektów — dane zmienialne mogą ulec zmianie na zewnątrz w wyniku działań wewnątrz funkcji. Typowy przykład takiego zachowania pokazuje Listing 7.24.

Listing 7.24. Parametry niezmienialne i zmienialne (tryb interaktywny)

```
1 >>> def dolacz(sekwencja1, sekwencja2):
  ...     sekwencja1 += sekwencja2
3 ...     return sekwencja1
  ...
5 >>> krotka = (1, 2, 3)
  >>> lista = [1, 2, 3]
7 >>> print dolacz(krotka, (5, 6, 7))
  (1, 2, 3, 5, 6, 7)
9 >>> print krotka
  (1, 2, 3)
11 >>> print dolacz(lista, [5, 6, 7])
  [1, 2, 3, 5, 6, 7]
13 >>> print lista
  [1, 2, 3, 5, 6, 7]
```

Funkcje bezparametrowe. Lista parametrów formalnych jest pusta, przykład widzimy powyżej, w Listingu 7.23. Nic nadzwyczajnego, znane jest to z wielu innych języków programowania.

„Zwyczajne" parametry. W definicji funkcji takie parametry formalne wymieniamy po przecinkach jedynie z nazwy (typów w Pythonie nie deklarujemy — także przy definiowaniu funkcji!) — patrz powyżej, Listing 7.24.

Parametry z wartościami domyślnymi. Funkcje w Pythonie mogą mieć parametry z wartościami domyślnymi. Można te parametry opuścić w wywołaniu funkcji (czyli są to parametry opcjonalne), a zostanie im przypisana wartość ustawiona w definicji. Parametry z wartościami domyślnymi nie mogą wystąpić w definicji przed parametrami bez nich (Listing 7.25, parametr `ile` ma wartość domyślną `1`).

Listing 7.25. Parametry domyślne (tryb interaktywny)

```
>>> import random
>>> def wybierz(lista, ile=1):
...     wynik = []
...     for i in range(ile):
...         wynik.append(random.choice(lista))
...     return wynik
...
>>> print wybierz([1, 2, 3, 4, 5])
[1]
>>> print wybierz([1, 2, 3, 4, 5])
[5]
>>> print wybierz([1, 2, 3, 4, 5], 1)
[5]
>>> print wybierz([1, 2, 3, 4, 5], 1)
[3]
>>> print wybierz([1, 2, 3, 4, 5], 3)
[2, 4, 3]
>>> print wybierz([1, 2, 3, 4, 5], 3)
[3, 1, 1]
>>> print wybierz([1, 2, 3, 4, 5], 10)
[2, 2, 3, 3, 1, 5, 2, 5, 2, 2]
```

Używając argumentów z wartościami domyślnymi, trzeba wiedzieć i pamiętać o tym, że są one tworzone/obliczane dokładnie raz, w momencie **definiowania** funkcji. Stąd pewne zaskakujące wyniki, dotyczące głównie danych zmienialnych, ale nie tylko. Listing 7.26 pokazuje dwie funkcje wraz z ich przykładowymi wywołaniami. Pierwsza z nich (`wybierz_el`) ma zwrócić element listy podanej w pierwszym parametrze o indeksie podanym w drugim parametrze; jeśli zaś ten drugi parametr jest opuszczony, to ma

być zwrócony element o indeksie wylosowanym (każdorazowo!) z liczb 0, 1, 2, 3. Natomiast funkcja `dolacz_el` ma dołączyć swój pierwszy argument do drugiego i zwrócić ów drugi po dołączeniu; w przypadku opuszczenia drugiego z argumentów należy przyjąć zań listę pustą.

Listing 7.26. Parametry domyślne z kłopotami (tryb interaktywny)

```
1 >>> import random
  >>> def wybierz_el(lista, ktory=random.randint(0, 3)):
3 ...     return lista[ktory]
  ...
5 >>> print wybierz_el([1, 2, 3, 4, 5], 0)
  1
7 >>> print wybierz_el([1, 2, 3, 4, 5], 2)
  3
9 >>> print wybierz_el([1, 2, 3, 4, 5])
  2
11 >>> print wybierz_el([1, 2, 3, 4, 5])
   2
13 >>> print wybierz_el([1, 2, 3, 4, 5])
   2
15 >>>
   >>> def dolacz_el(co, gdzie=[]):
17 ...     gdzie.append(co)
   ...     return gdzie
19 ...
   >>> print dolacz_el(4, [1, 2, 3])
21 [1, 2, 3, 4]
   >>> print dolacz_el(5, [])
23 [5]
   >>> print dolacz_el(6, [])
25 [6]
   >>> print dolacz_el(7)
27 [7]
   >>> print dolacz_el(8)
29 [7, 8]
   >>> print dolacz_el(9)
31 [7, 8, 9]
```

Niestety, nasze funkcje działają niezgodnie z oczekiwaniami (linie 9–14, gdzie wybierany jest zawsze ten sam element, a nie losowy; oraz linie 26–31, gdzie widać, że kolejne elementy nie są dołączane do pustej listy). Dlaczego tak się dzieje? W pierwszym przypadku wartość funkcji `random.randint(0, 3)` jest obliczana tylko raz, w linii 2, a potem ta wartość jest już stale używana jako wartość domyślna parametru `ktory`.

Przypadek drugi jest nieco bardziej skomplikowany, bowiem lista jest obiektem zmienialnym. W związku z tym, oraz dlatego, że wartość domyślna tworzona jest tylko raz, tylko pierwsze wywołanie funkcji `dolacz_el` bez

drugiego argumentu (linia 26) daje spodziewany wynik. Wartość domyślna tak naprawdę się nie zmienia, ale jest ona przecież (jak wszelkie zmienne w Pythonie) nie listą, lecz referencją do niej. Referencja się nie zmienia, ale zmienia się zawartość listy, na którą wskazuje. Stąd lista będąca wartością domyślną rośnie z każdym wywołaniem bez drugiego parametru...

Typowe przykładowe rozwiązania usuwające powyższe problemy pokazuje Listing 7.27.

Listing 7.27. Parametry domyślne z poprawionymi kłopotami

```
1 import random
  def wybierz_el(lista, ktory=None):
3   if ktory == None:
      ktory = random.randint(0, 3)
5   return lista[ktory]

7 def dolacz_el(co, gdzie=None):
    if gdzie == None:
9     return [co]
    gdzie.append(co)
11  return gdzie
```

Parametry pozycyjne. Parametry te są oznaczane jednym identyfikatorem poprzedzonym symbolem `*` (na przykład przez `*a`, gdzie `a` może być dowolnym identyfikatorem) i w definicji funkcji nie mogą wystąpić przed parametrami zwykłymi i parametrami z wartościami domyślnymi. Funkcja zdefiniowana z parametrami pozycyjnymi może przyjąć dowolną liczbę parametrów wywołania i wszelkie nadmiarowe, które „nie mieszczą" się w standardowych parametrach formalnych są przekazywane jako kolejne elementy krotki dostępnej pod nazwą podaną po gwiazdce. Prosty przykład użycia parametrów pozycyjnych w działaniu pokazuje Listing 7.28, który powinien rozwiać wszelkie wątpliwości.

Listing 7.28. Parametry zwykłe, domyślne i pozycyjne (tryb interaktywny)

```
1 >>> def f(a, b=0, *c):
  ...     print a, b, c
3 ...
  >>> f(10)
5 10 0 ()
  >>> f(10, 12)
7 10 12 ()
  >>> f(10, 12, 15)
9 10 12 (15,)
  >>> f(10, 12, 15, 18)
11 10 12 (15, 18)
```

```
>>> f(10, 12, 15, 18, 100)
10 12 (15, 18, 100)
```

Parametry nazwane. Jeśli parametry nazwane[18] występują w definicji funkcji, to muszą być one ostatnie wśród parametrów formalnych i — podobnie jak parametry pozycyjne — oznaczane są jednym (dowolnym) identyfikatorem, tym razem poprzedzonym dwiema gwiazdkami: `**k`. Taki parametr formalny w wywołaniu staje się słownikiem, do którego trafiają wszystkie nazwane parametry aktualne wywołania, które nie mają swoich odpowiedników w nazwach parametrów formalnych. Przykład ich użycia odłożymy do następnego podrozdziału (Podrozdział 7.4.2, Listing 7.31) omawiającego parametry wywołania.

7.4.2. Parametry aktualne

Parametry aktualne funkcji (czyli jej wywołania) dzielą się na parametry nazwane i pozycyjne, których każdy rodzaj można przekazać na dwa sposoby.

Wykaz parametrów pozycyjnych. W wywołaniu funkcji najzwyklejszym sposobem przekazania parametrów aktualnych jest podanie po przecinkach wyrażeń, które są przypisywane kolejnym parametrom formalnym — tu (jak w wielu innych językach) o wzajemnym przyporządkowaniu parametrów formalnych i aktualnych decyduje jedynie kolejność. Do tej pory stosowaliśmy takie właśnie przekazywanie parametrów — Listingi 7.24–7.28.

Wykaz parametrów nazwanych. Te nie mogą wystąpić przed wykazem parametrów pozycyjnych. Podaje się je w formie analogicznej do parametrów formalnych z wartościami domyślnymi: każdy taki parametr podajemy w formie `n=w`, gdzie `n` jest nazwą odpowiedniego parametru formalnego, natomiast `w` jest wyrażeniem, którego wartość chcemy przekazać danemu parametrowi. Kolejność w tym wykazie nie gra roli — jedynie podane nazwy parametrów formalnych — Listing 7.29.

Listing 7.29. Wykaz parametrów aktualnych nazwanych (tryb interaktywny)

```
>>> def f(a, b, c):
...     print a, b, c
...
```

[18] Funkcjonuje też pojęcie *parametry słów kluczowych* (ang. *keyword arguments*) ale niefortunnie kojarzy się ono ze słowami kluczowymi języka, więc my tu go nie używamy.

```
>>> f(1, 2, 3)
1 2 3
>>> f(b=1, c=2, a=3)
3 1 2
>>> f(1, c=2, b=3)
1 3 2
```

Parametry pozycyjne jako sekwencja. Parametry pozycyjne mogą być podane także jako dowolna sekwencja (zwykle lista lub krotka) poprzedzona symbolem * (o ile nie ma wykazu parametrów nazwanych). Jeśli w liście parametrów aktualnych podamy taką sekwencję poprzedzoną gwiazdką, to jej elementy zostaną przypisane kolejnym (nieobsłużonym przez wcześniejsze parametry aktualne) parametrom formalnym — Listing 7.30.

Listing 7.30. Parametry pozycyjne jako krotka (tryb interaktywny)

```
>>> def f(a, b, c):
...     print a, b, c
...
>>> k = (10, 20)
>>> l = (100, 200, 300)
>>> f(1, *k)
1 10 20
>>> f(*l)
100 200 300
>>> f(*reversed(l))
300 200 100
>>> f(1, *reversed(k))
1 20 10
```

Parametry nazwane jako słownik. W końcu możemy użyć gotowego słownika, który w liście parametrów oznaczymy ** (zawsze na końcu listy parametrów). Dane z tego słownika traktowane są jak parametry nazwane (kluczami słownika) o wartościach ze słownika — Listing 7.31.

Listing 7.31. Parametry nazwane (tryb interaktywny)

```
>>> k = (1, 2, 3)
>>> s1 = {'a': 100, 'b': 200, 'c': 300}
>>> s2 = {'a': 100, 'b': 200, 'c': 300, 'd': 400}
>>> s3 = {'a': 100, 'c': 300, 'd': 400}
>>> s4 = {'c': 300, 'd': 400}
>>> def f(a, b, c):
...     print 'a:', a, 'b:', b, 'c:', c
...
>>> def g(a, b=10, *poz, **naz):
```

```
    ...     print 'a:', a, 'b:', b, 'poz:', poz, 'naz:', naz
11  ...
    >>> f(*k)
13  a: 1 b: 2 c: 3
    >>> f(**s1)
15  a: 100 b: 200 c: 300
    >>> g(*k)
17  a: 1 b: 2 poz: (3,) naz: {}
    >>> g(**s1)
19  a: 100 b: 200 poz: () naz: {'c': 300}
    >>> g(**s2)
21  a: 100 b: 200 poz: () naz: {'c': 300, 'd': 400}
    >>> g(**s3)
23  a: 100 b: 10 poz: () naz: {'c': 300, 'd': 400}
    >>> g(1, **s4)
25  a: 1 b: 10 poz: () naz: {'c': 300, 'd': 400}
    >>> g(1, 2, **s4)
27  a: 1 b: 2 poz: () naz: {'c': 300, 'd': 400}
    >>> g(*k, **s4)
29  a: 1 b: 2 poz: (3,) naz: {'c': 300, 'd': 400}
    >>> g(1, *k, **s4)
31  a: 1 b: 1 poz: (2, 3) naz: {'c': 300, 'd': 400}
```

7.4.3. Zakres widoczności

W Pythonie zakres widoczności obiektów jest dynamiczny (strona 43, Podrozdział 3.2.3). Ma to znaczenie dla semantyki i działania podprogramów, bowiem w Pythonie wszystko — także funkcje — jest obiektem i jest powoływane do życia w konkretnym momencie **działania** programu. W przypadku funkcji jest to moment jej zdefiniowania za pomocą instrukcji `def`, która jest **całkiem zwyczajną instrukcją** — o czym świadczyć może program z Listingu 7.32, w którym w zależności od odpowiedzi użytkownika definiowana jest inna treść funkcji `f`.

Listing 7.32. Dynamiczne definiowanie funkcji i zakres widoczności zmiennych

```
1  def g():
       return f(odp)
3  def h():
       odp = 16.0
5      return f(odp)
   odp = float(raw_input("Podaj liczbe: "))
7  if odp < 0.0:
       def f(x):
9          return x**2.0
   else:
11     def f(x):
```

```
        return x**0.5
13 print g(), h(), odp
```

Więcej na temat funkcji w Pythonie dalej, w Podrozdziale 7.6, o programowaniu funkcyjnym.

7.5. Obiektowość w Pythonie

Jak już kilkakrotnie podkreślaliśmy, w Pythonie wszystko (liczby, listy, funkcje, moduły...) jest obiektem, tak więc Python jest językiem całkowicie obiektowym. Obiektowość w Pythonie ma jednak pewne swoiste cechy, które zdecydowanie odróżniają ten język od innych obiektowych, nieco bardziej popularnych (jak Java czy C++).

7.5.1. Stary i nowy styl klas

Oczywiście centralnym pojęciem jest *klasa* (także będąca obiektem!), czyli typ, który definiujemy za pomocą słowa instrukcji `class`, której zawartością jest przede wszystkim zestaw definicji metod (i, rzadziej, własności) klasowych. Jeśli klasa ma być pusta, musimy w jej definicji umieścić instrukcję pustą `pass`. Definicja klasy może zawierać także (analogicznie do definicji funkcji) komentarz dokumentujący.

Klasy mogą być definiowane w dwóch stylach — starym (klasy samodzielne, klasy bez korzenia) i nowym. Klasa w nowym stylu zawsze dziedziczy[19] (ale nie koniecznie bezpośrednio) po klasie standardowej `object` — jeśli nie, to jest w starym stylu (Listing 7.33). Klasy w starym stylu są zaszłością z poprzednich wersji zachowaną dla kompatybilności. Zalecane jest jednak ich niestosowanie w nowych programach i my także będziemy posługiwać się tylko nowym stylem.

Listing 7.33. Style definiowania klas

```
1 class PustaKlasaWStarymStylu:
    pass
3
  class PustaKlasaWNowymStylu(object):
5   pass
```

[19] W Pythonie dziedziczenie wielokrotne jest dopuszczalne — na przykład nagłówek definicji klasy `class A(B, C, D):` mówi, że klasa `A` dziedziczy po trzech klasach: `B`, `C` oraz `D`. Ponadto klasa pochodna jest w Pythonie formalnie podtypem klasy bazowej.

7.5.2. Jawność

Zaskakujące dla wielu programistów zaczynających programowanie w Pythonie, ale znających już reguły programowania obiektowego w innych językach jest brak kwalifikatorów dostępu (jak `public`, `private`, `protected`). Wszystkie pola i metody wszystkich obiektów są publiczne[20]!

Projektanci Pythona liczą tu (całkiem świadomie [79]) na dojrzałość programistów — tak projektujących klasy, jak i korzystających z cudzych klas. Ci pierwsi powinni dobrze dokumentować swoje klasy, ci drudzy — czytać i stosować się do dokumentacji. Przy założeniu takiej dojrzałości jasne jest, że żadne kwalifikatory dostępu nie są potrzebne, bo nikt nic nie popsuje...

7.5.3. Tworzenie obiektu

Obiekt tworzymy wywołując klasę (jak funkcję) z odpowiednimi argumentami (patrz metoda `__init__`, niżej), na przykład `PustaKlasaWNowymStylu()` — takie wywołanie za każdym razem tworzy nową instancję klasy (czyli obiekt).

7.5.4. Niedeklarowanie pól

Pola obiektów są w Pythonie traktowane jak wszystkie inne zmienne, a więc nie deklaruje się ich wcześniej, tylko podstawia, gdy trzeba. Listing 7.34 przedstawia korzystającą z tego dość luźną implementację rekordów znanych z innych języków programowania (bardziej ścisła implementacja rekordów w Listingu 7.39 na stronie 167).

Listing 7.34. Pola obiektów (tryb interaktywny)

```
>>> class Rekord(object):
...     pass
...
>>> a = Rekord()
>>> a.imie = 'Jan'
>>> a.wiek = 28
```

7.5.5. Metody zwyczajne; metody i własności specjalne

Funkcje definiowane w obrębie klasy muszą mieć zadeklarowany przynajmniej jeden parametr (zwyczajowo nazywany `self`, ale nazwa może być dowolna). Funkcje te mogą być wywoływane na rzecz każdego obiektu tej

[20] Właściwie są pewne metody ochrony różnych szczegółów implementacji klasy przed jej użytkownikiem, ale są one nienaturalne i niezbyt wygodne.

klasy przez zwyczajowy zapis z kropką, pomija się jednak w tych wywołaniach ów pierwszy parametr, którym jest zawsze automatycznie przekazywana referencja do wywołującego obiektu.

Obiekty i klasy w Pythonie mogą mieć określone specjalne zachowanie dzięki zdefiniowaniu specjalnych metod[21], z których najważniejszą jest `__init__`. Metoda ta pełni rolę *inicjalizatora*[22], czyli funkcji wywoływanej natychmiast po utworzeniu obiektu — zwykle w celu ustawienia domyślnych (czy też podanych w argumentach inicjalizatora) wartości pól obiektu.

Inne specjalne metody, o których warto pamiętać to `__str__` oraz `__repr__`, które mają za zadanie zwracać napisową reprezentację obiektu (pierwsza w formie czytelnej dla człowieka; druga w formie możliwej do odtworzenia w Pythonie, a więc zwykle jako zapis wywołania konstruktora). Metody te są wykorzystywane między innymi przez standardowe funkcje `str` i `repr` oraz instrukcję `print`.

Dla ilustracji rozważmy klasę obiektów przechowujących temperaturę i przeliczających ją automatycznie pomiędzy skalami Kelvina, Celsjusza, Fahrenheita (pomysł za [33]) — Listing 7.35.

Listing 7.35. Klasa przeliczająca temperatury

```
class Temperatura(object):
    """Klasa do automatycznego przeliczania miedzy
    skalami temperatur."""
    wspolczynniki = {
        'c': (1.0, 0.0, -273.15),
        'k': (1.0, 0.0, 0.0),
        'f': (1.8, -273.15, 32.0)
    }
    def __init__(self, ile=0.0, skala='k'):
        """Inicjalizacja jak metoda 'ze_skali',
        ale z domyslnym zerem absolutnym w kelwinach."""
        self.ze_skali(ile, skala)
    def ze_skali(self, ile, skala):
        """Przypisuje aktualnie przechowywanej
        temperaturze nowa wartosc podana w danej skali
        (musi to byc jedna z podanych we
        'wspolczynnikach')."""
        w = self.wspolczynniki[skala]
```

[21] Wszystkie metody specjalne i pola specjalne mają nazwy rozpoczynające się od dwóch znaków podkreślenia i kończące dwoma znakami podkreślenia.

[22] Stosowana w tym kontekście nazwa *konstruktor* nie do końca jest w Pythonie poprawna, bo funkcja `__init__` dostaje już skonstruowany obiekt i na nim wykonuje pewne czynności inicjujące. Ma to przede wszystkim tę konsekwencję, że gdy wykonywana jest metoda `__init__` dla klasy potomnej, a chcemy, by wykonała się wcześniej inicjalizacja zdefiniowana w klasie bazowej, musimy zrobić to ręcznie, zawierając w definicji `__init__` dla klasy potomnej wywołanie podobne do:
`KlasaBazowa.__init__(self, argumenty)`

```
        self.kelwiny = (ile - w[2]) / w[0] - w[1]
    def na_skale(self, skala):
        """Zwraca akualnie przechowywana temperature
        przeliczona na podana skale (musi to byc jedna
        ze skal podanych we 'wspolczynnikach')."""
        w = self.wspolczynniki[skala]
        return (self.kelwiny + w[1]) * w[0] + w[2]
    def __str__(self):
        """Zwraca temperature (w kelwinach) jako czytelny
        napis."""
        return "%s K" % (self.kelwiny,)
    def __repr__(self):
        """Zwraca Pythonowa reprezentacje przechowywanej
        temperatury."""
        return "Temperatura(%s, 'k')" % (self.kelwiny,)
```

Zakładając, że kod z Listingu 7.35 mamy w pliku `temperatura.py`, możemy teraz wypróbować go — na przykład interaktywnie (Listing 7.36).

Listing 7.36. Test klasy przeliczającej temperatury (tryb interaktywny)

```
>>> from temperatura import Temperatura
>>> t = Temperatura()
>>> t.ze_skali(100, 'c')
>>> print t.na_skale('k')
373.15
>>> print t.na_skale('f')
212.0
>>> print t
373.15 K
>>> t
Temperatura(373.15, 'k')
>>> print t.__doc__
Klasa do automatycznego przeliczania miedzy
    skalami temperatur.
>>> print t.na_skale.__doc__
Zwraca aktualnie przechowywana temperature
        przeliczona na podana skale (musi to byc jedna
        ze skal podanych we 'wspolczynnikach').
```

W dwóch ostatnich poleceniach widzimy, że mamy prosty dostęp do komentarza dokumentującego przez pole specjalne `__doc__`, a ponadto ostatnie polecenie dowodzi, że funkcje/metody też są obiektami i mają swoje własności (tutaj właśnie `__doc__`).

Inne ważne metody specjalne to metody definiujące operatory — dzięki nim możemy zdefiniować (i przeciążyć) operatory działające na obiektach definiowanej klasy.

Na przykład, spróbujmy zdefiniować klasę `Wym` realizującą operacje na

liczbach wymiernych w postaci ułamków zwykłych — Listing 7.37. Z operacji arytmetycznych ograniczymy się jedynie do dodawania[23].

Listing 7.37. Klasa liczb wymiernych

```
class Wym(object):
    def __init__(self, licznik=0, mianownik=None):
        if isinstance(licznik, Wym):
            if mianownik == None:
                self.l = licznik.l
                self.m = licznik.m
            else:
                raise TypeError(
"""dopuszczalna inicjalizacja:
Wym(),
Wym(wymierna),
Wym(calkowita),
Wym(calkowita, calkowita)""")
        else:
            if mianownik == None:
                mianownik = 1
            self.l = int(licznik)
            self.m = int(mianownik)
        self.normalizuj()
    def normalizuj(self):
        if self.m == 0:
            raise ZeroDivisionError('mianownik zerowy')
        if self.l == 0:
            self.m = 1
        if self.m < 0:
            self.l = -self.l
            self.m = -self.m
        a = abs(self.l)
        b = abs(self.m)
        while b:
            a, b = b, a % b
        self.l /= a
        self.m /= a
    def __str__(self):
        return "%s/%s" % (self.l, self.m)
    def __repr__(self):
        return "Wym(%s, %s)" % (self.l, self.m)
    def podloga(self):
        return self.l//self.m
    def zmiennop(self):
        return float(self.l)/float(self.m)
    def __add__(self, inny):
        inny = Wym(inny)
        return Wym(self.l*inny.m+self.m*inny.l,
                   self.m*inny.m)
```

[23] Pozostałe operacje: patrz zadanie 9 na końcu niniejszego rozdziału, strona 181.

```
46      def __radd__(self, inny):
            return self.__add__(inny)
```

Zrezygnowaliśmy z umieszczania tu komentarzy dokumentujących (normalnie mocno zalecanych), bowiem najważniejsze punkty z powyższego kodu omówimy tutaj, zanim przetestujemy działanie klasy w praktyce (Listing 7.38).

— Metoda `__init__` (wiersz 2–19) jest tak długa, bo musimy wziąć w niej pod uwagę trzy sposoby wywołania. Chcemy bowiem mieć różne możliwości inicjowania obiektu (bez parametrów [argumenty domyślne nam to zapewniają], jedną liczbą wymierną [wiersze 3–13], jedną liczbą całkowitą [wiersze 15–16] albo dwiema liczbami całkowitymi [wiersze 17–18]). Dzięki temu mamy jedną metodę, którą możemy wywoływać z różną liczbą i typem argumentów[24].
— Wiersz 3 sprawdza czy podany parametr należy do danego typu (tu: do `Wym`).
— Wiersze 8–13 oraz 22 wyrzucają standardowy wyjątek (ale z własnym komunikatem) w przypadku problemów.
— Metoda `normalizuj` (wiersze 20–33) pełni funkcje pomocnicze i doprowadza liczbę do zapisu zestandaryzowanego (mianownik dodatni, ułamek skrócony, jeśli licznik zerowy, to mianownik równy 1).
— Wiersz 28–31 to oczywiście algorytm Euklidesa. Obywa się on jednak w Pythonie bez dodatkowej zmiennej pomocniczej, bo można tutaj zamienić dane bez udziału tymczasowej przechowalni (linia 31).
— Wiersze 34–37 definiują specjalne metody znane już z poprzedniego przykładu (Listing 7.35).
— Wiersze 38–41 zawierają definicje metod konwertujących naszą liczbę wymierną na liczbę całkowitą (`podloga`, przez ucięcie części ułamkowej) oraz na liczbę zmiennoprzecinkową (`zmiennop`, przez zmiennoprzecinkowe podzielenie licznika przez mianownik).
— W końcu wiersze 42–47 definiują dwie nowe metody umożliwiające używanie operatora `+` na danych typu `Wym`. Jak to się dzieje? Otóż, wyrażenie `a+b` jest w Pythonie równoważne wyrażeniu `a.__add__(b)` lub wyrażeniu `b.__radd__(a)`. Dlaczego dwa przypadki? Pierwsza z tych metod

[24] Bo właściwie funkcji w Pythonie się nie przeciąża w klasycznym tego słowa znaczeniu (strona 4.1.4). Oczywiście, mogą być różne metody nazywające się identycznie w różnych klasach/obiektach, ale nie są one rozpoznawane po sygnaturze, lecz jedynie po wywołującym je obiekcie — i są tu traktowane tak samo, jak pola o tych samych nazwach w różnych obiektach. Jeśli natomiast w danym kontekście (module, pliku, klasie, obiekcie) chcemy mieć pod tą samą nazwą podprogramu różne czynności wykonywane w zależności od liczby i typów argumentów, to musimy zrobić to „ręcznie” — jak `__init__` w kodzie 7.37. Oczywiście dla użytkownika danej metody jej wywołanie wygląda, jak przy klasycznym polimorfizmie, jedynie jej twórca musi zastosować nieco inne podejście.

jest używana domyślnie, ale jeśli nie jest zdefiniowana lub powoduje błąd, Python próbuje wykonać drugą z nich (jednakże z odwróconymi rolami operandów). Ponieważ chcemy, żeby nasze liczby wymierne mogły sumować się bez problemu ze standardowymi liczbami całkowitymi, musimy zdefiniować zarówno metodę `__add__`, która potrafi do danej liczby wymiernej (lewy operand operatora `+`) dodać inną wymierną lub całkowitą (prawy operand), jak i metodę `__radd__`, która potrafi dodać wymierną (prawy operand operatora `+`) do liczby całkowitej (lewy operand). W naszym przypadku — ponieważ dodawanie liczb jest przemienne — definiujemy drugą z tych metod za pomocą pierwszej (wiersze 46–47).

Listing 7.38 pokazuje, jak działają nasze liczby wymierne w praktyce (przy założeniu, że plik z powyższą klasą ma nazwę `wymierne.py`).

Listing 7.38. Test liczb wymiernych (tryb interaktywny)

```
>>> from wymierne import Wym
>>> w1 = Wym()
>>> w2 = Wym(-23)
>>> w3 = Wym(130, 7)
>>> w4 = Wym(w3)
>>> w5 = w3
>>> w4 is w3
False
>>> w5 is w3
True
>>> w6 = w2+w3
>>> print w1, w2, w3, w4, w5, w6
0/1 -23/1 130/7 130/7 130/7 -31/7
>>> print w6+5
4/7
>>> print 5+w6
4/7
>>> w1
Wym(0, 1)
>>> w2
Wym(-23, 1)
>>> w6
Wym(-31, 7)
>>> 5+w6
Wym(4, 7)
>>> print w2.podloga(), w3.podloga(), w6.podloga()
-23 18 -5
>>> print w2.zmiennop(), w3.zmiennop(), w6.zmiennop()
-23.0 18.5714285714 -4.42857142857
```

Jeszcze jeden przykład — obiecana wcześniej implementacja „prawdzi-

wych" rekordów (niepozwalających na tworzenie nowych pól, ani na usuwanie już istniejących): Listingi 7.39 oraz 7.40.

Listing 7.39. Moduł `rekordy.py` implementujący tradycyjne rekordy

```
class Rekord(object):
    def __init__(self, *lista_pol, **slownik_pol):
        for n in lista_pol:
            object.__setattr__(self, n, None)
        for (n, w) in slownik_pol.items():
            object.__setattr__(self, n, w)
    def __delattr__(self, nazwa):
        raise AttributeError('nie mozna usuwac pol')
    def __setattr__(self, nazwa, wart):
        if nazwa in self.__dict__:
            object.__setattr__(self, nazwa, wart)
        else:
            raise AttributeError('nie ma takiego pola')
    def __repr__(self):
        r=[]
        for (n, w) in self.__dict__.items():
            r += ['%s=%s' % (n, repr(w))]
        return 'Rekord(%s)' % (', '.join(r),)
```

Listing 7.40. Test modułu `rekordy.py` (tryb interaktywny)

```
>>> from rekordy import Rekord
>>> r1 = Rekord('imie', 'nazwisko', 'wiek')
>>> r1
Rekord(imie=None, nazwisko=None, wiek=None)
>>> r1.imie = 'Jan'
>>> r1.nazwisko = 'Kowalski'
>>> r1.wiek = 39
>>> r1.zonaty = True
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rekordy.py", line 13, in __setattr__
    raise AttributeError('nie ma takiego pola')
AttributeError: nie ma takiego pola
>>> del r1.wiek
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rekordy.py", line 8, in __delattr__
    raise AttributeError('nie mozna usuwac pol')
AttributeError: nie mozna usuwac pol
>>> r1
Rekord(imie='Jan', nazwisko='Kowalski', wiek=39)
>>> r2 = Rekord(x=3.5, y=10.0)
>>> r2
Rekord(y=10.0, x=3.5)
```

```
>>> print r2.x, r2.y
3.5 10.0
```

Co nowego znajdujemy w powyższych Listingach?

— Metoda `__delattr__` (definicja w wierszach 7–8 Listingu 7.39) jest wywoływana, gdy ma nastąpić usunięcie pola z obiektu, natomiast metoda `__setattr__` (wiersze 9–13 Listingu 7.39) — gdy ma nastąpić przypisanie do pola obiektu.
— W wierszu 11 Listingu 7.39 użyta jest metoda z klasy bazowej, bo inaczej doszłoby do nieskończonej rekurencji.
— Pole specjalne `__dict__` (wiersze 10, 16 Listingu 7.39) jest słownikiem wszystkich aktualnie istniejących pól obiektu (wraz z ich wartościami).
— Dzięki zastosowaniu „gwiazdek" w definicji metody `__init__` (wiersz 2 Listingu 7.39) mamy możliwość różnorodnego definiowania startowego zestawu pól rekordu (co widać w wierszach 2 i 22 Listingu 7.40).
— Metoda `items` słowników (wiersze 5 i 16 Listingu 7.39) zwraca sekwencję krotek `(klucz, wartość)` dla elementów słownika.
— Metoda napisowa `join` (wiersz 18 Listingu 7.39) łączy elementy listy w jeden napis z separatorem podanym jako `self` (i jest to bardziej efektywne od ich ręcznej konkatenacji z uwagi na niezmienialność napisów).
— Instrukcja `del` (wiersz 14 Listingu 7.40) usuwa daną — może to być obiekt, pole obiektu, element listy lub słownika itp.

7.5.6. Zgodność sygnatur

Jak już wspomnieliśmy poprzednio, ścisła kontrola typów w Pythonie polega przede wszystkim na tym, że mało jest automatycznych konwersji pomiędzy typami. Jednakże z drugiej strony w Pythonie panuje *polimorfizm oparty na sygnaturach* (w Pythonie równoważnie: *polimorfizm oparty na protokołach*, czyli na zestawie dostępnych dla obiektu metod; znany też jako *kacze typowanie*[25]). Oznacza to, że jeśli przygotujemy na przykład funkcję, która nie sprawdza typu swojego argumentu, ale po prostu stosuje do niego pewne metody, to możemy ją wywołać z argumentem każdego typu mającego takie metody!

Jest to odmiana *programowania generycznego* znanego także z szablonów w C++, pakietów generycznych w Adzie czy typów generycznych w Javie — tyle tylko, że w Pythonie wszystko może być takim szablonem bez specjalnej dodatkowej składni.

[25] Pojęcie to (ang. *duck typing*) wzięło się od powiedzenia „jeśli widzisz coś, co chodzi jak kaczka, gdacze jak kaczka, pływa jak kaczka, traktujesz to jak kaczkę" przypisywanego w tej czy innej formie Jamesowi Whitcombowi Rileyowi — poecie i pisarzowi z przełomu XIX i XX wieku [73].

Listing 7.41 wyjaśnia to zagadnienie na przykładzie. Definiujemy tu funkcję `razem`, która zbiera razem swoje argumenty w jeden za pomocą operatora `+=`. Działać będzie więc dla każdego typu, dla którego zdefiniowane jest dodawanie — nie tylko dla typów standardowych, ale także dla naszego typu `Wym` z Listingu 7.37.

Listing 7.41. Polimorfizm oparty na sygnaturach (tryb interaktywny)

```
>>> def razem(*ciag):
...     wynik = ciag[0]
...     for el in ciag[1:]:
...         wynik += el
...     return wynik
...
>>> razem(1, 2, 3, 10, 500)
516
>>> razem('1', '2', '3', '10', '500')
'12310500'
>>> razem(('1', '2'), ('3', '10', '500'), ())
('1', '2', '3', '10', '500')
>>> from wymierne import Wym
>>> razem(Wym(1, 2), Wym(10, 23), Wym(7,8), Wym(-1))
Wym(149, 184)
```

7.5.7. Programowanie refleksyjne

Refleksyjność (czyli zdolność programu do samoinspekcji i samomodyfikacji) Pythona jest ściśle związana z jego obiektowością — bo narzędzia refleksyjne są atrybutami obiektów. Dobrym i praktycznym przykładem szerokiego i zaawansowanego zastosowania mechanizmu refleksji w Pythonie jest PLY [82] — implementacja Lexa i Yacca [41] w Pythonie. Sami także używaliśmy tego mechanizmu (na przykład `__doc__`, `__dict__`, a w Listingu 7.42 pojawia się funkcja `dir`). Jeszcze jeden przykład przedstawia Listing 7.42.

Listing 7.42. Zastosowanie mechanizmu refleksji

```
def pom(ob):
    """Zwraca (w formie napisu) dokumentacje
    obiektu oraz wykaz jego metod wraz z ich
    dokumentacja. Przyklady uzycia:
    print pom(pom)
    print pom([])
    print pom(Wym)
    print pom(wymierne)
    print pom(temperatura)
    print pom(temperatura.Temperatura)"""
```

```
    funkcje = []
    for x in dir(ob):
        el = getattr(ob, x)
        if callable(el):
            funkcje.append((x, el))
    funkcje.sort()
    wynik=[ob.__doc__ or '']
    for (x, el) in funkcje:
        wynik.append('------>%s:%s' % (x, el.__doc__ or ''))
    return '\n'.join(wynik)
```

7.5.8. Deskryptory

Jedną z ciekawych cech programowania obiektowego w Pythonie jest możliwość definiowania *deskryptorów* [19], czyli klas dla atrybutów (pól) obiektów mających specjalny dostęp. Przykład klasy deskryptorowej `PoleSledzone` i jej wykorzystanie w pewnej klasie pokazują Listingi 7.43 i 7.44. Oprócz pokazanych tu metod `__get__` oraz `__set__` deskryptory mogą także definiować metodę `__delete__`. Nasza definicja deskryptora zakłada, że obiekty klasy z niego korzystającej mają pole `opis`. Klasę `K` natomiast definiujemy tak, by śledzone były pola `a` oraz `c`, ale już nie pole `b`.

Listing 7.43. Definicja prostego deskryptora (plik `deskryptory.py`)

```
class PoleSledzone(object):
    def __init__(self, opis, wart_pocz=None):
        self.w = wart_pocz
        self.o = opis
    def __get__(self, ob, typ):
        print 'GET: %s==%s (w: %s)' % (self.o, self.w, ob.opis)
        return self.w
    def __set__(self, ob, wart):
        print 'SET: %s=%s (w: %s)' % (self.o, wart, ob.opis)
        self.w = wart
```

Listing 7.44. Wykorzystanie prostego deskryptora (tryb interaktywny)

```
>>> from deskryptory import PoleSledzone
>>> class K(object):
...     def __init__(self, opis='anonimowy obiekt klasy K'):
...         self.opis = opis
...     a = PoleSledzone('pole a')
...     b = 100
...     c = PoleSledzone('pole c')
...
```

```
>>> k = K()
>>> k1 = K('k1')
>>> k2 = K('k2')
>>> k.a = 100
SET: pole a=100 (w anonimowy obiekt klasy K)
>>> k2.c = 3000
SET: pole c=3000 (w k2)
>>> print k.a+k1.b+k2.c
GET: pole a==100 (w anonimowy obiekt klasy K)
GET: pole c==3000 (w k2)
3200
>>> for i in range(5):
...     k2.a, k2.b = i, i
...
SET: pole a=0 (w k2)
SET: pole a=1 (w k2)
SET: pole a=2 (w k2)
SET: pole a=3 (w k2)
SET: pole a=4 (w k2)
```

7.6. Programowanie funkcyjne w Pythonie

Python oferuje także właściwie w pełni rozwinięty paradygmat funkcyjny[26] — włącznie z takimi jego cechami, jak funkcje działające na funkcjach, elementy notacji lambda, leniwe wartościowanie, struktury potencjalnie nieskończone itp. Poniżej podajemy ogólny przegląd tych cech (z przykładami), głównie na podstawie [27] — i tam też zachęcamy uzupełnić i poszerzyć wiedzę na temat programowania funkcyjnego w Pythonie.

7.6.1. Funkcje jako wartości pierwszego rzędu; domknięcia

Jak wszystko w Pythonie, funkcje także są obiektami. Umożliwia to programowanie funkcyjne, bo funkcje dzięki temu w naturalny sposób mogą być zarówno danymi do innych funkcji, jak i wynikami funkcji (takie funkcje zdefiniowane w funkcji i zwrócone na zewnątrz to *domknięcia*). Żeby zaprezentować przykład takiego podejścia zdefiniujemy kilka funkcji operujących na funkcjach i prześledzimy ich działanie (Listing 7.45).

Listing 7.45. Funkcje jako wartości pierwszego rzędu

```
import pprint
import math
import operator
```

[26] Oczywiście nie jest to czysta funkcyjność, bo Python jest jednocześnie imperatywny.

```
5 def odwzoruj(fun, sekw):
      wynik = []
7     for el in sekw:
          wynik.append(fun(el))
9     return wynik

11 pprint.pprint(odwzoruj(math.sqrt, range(17)))

13 def redukcja(fun, sekw, neutr):
      wynik = neutr
15    for el in sekw:
          wynik = fun(wynik, el)
17    return wynik

19 print redukcja(operator.add, ['qw', 'ert', 'y'], '')
   silnia1 = lambda n: redukcja(
21                          operator.mul,
                            range(1, n+1), 1)
23 print silnia1(6)
   silnia2 = lambda n: redukcja(
25                          (lambda x, y: x*y),
                            range(1, n+1), 1)
27 print silnia2(6)

29 def zlozenie1(f, g):
      def fg(*a, **k):
31        return f(g(*a, **k))
      return fg
33 def zlozenie2(f, g):
      return lambda *a, **k: f(g(*a, *kk))
35 zlozenie3 = lambda f, g: (lambda *a, **k: f(g(*a, *kk)))

37 sin_w_stopniach = zlozenie1(math.sin, math.radians)
   print sin_w_stopniach
39 print sin_w_stopniach(90)
   pprint.pprint(odwzoruj(sin_w_stopniach, range(0, 91, 5)))
41 pprint.pprint(odwzoruj(
                    zlozenie1(math.sin, math.radians),
43                  range(0, 91, 5)))

45 def samozlozenie(f, n):
      def fn(x):
47        wynik = x
          for i in range(n):
49            wynik = f(wynik)
          return wynik
51    return fn

53 def f_wykl(m):
      def fw(x):
```

```
55            return x**m
        return fw
57 print samozlozenie(f_wykl(2), 10)(3)
```

Miejsca kodu, na których warto się skupić:

— Linie 1–3 importują standardowe moduły.

— Funkcja `odwzoruj` zwraca listę równoliczną z drugim argumentem, ale taką, że każdy element listy wynikowej jest wynikiem działania funkcji danej w pierwszym argumencie na odpowiednim elemencie drugiego argumentu (analogicznie do `map` — strona 95 w Podrozdziale 23). W wierszu 11 widać działanie tej funkcji (wyświetlana jest lista pierwiastków kwadratowych liczb całkowitych od 0 do 16).

— Funkcja `redukcja` zwraca wynik zastosowania działania podanego w pierwszym argumencie do wszystkich elementów sekwencji będącej drugim argumentem. Obliczanie zaczyna się od elementu neutralnego (trzeci argument), do którego „dokładane" są poszczególne elementy sekwencji. Wiersz 19 pokazuje działanie tej funkcji.

— Wiersze 20–27 pokazują dwie równoważne definicje silni za pomocą redukcji[27]. W obu występuje funkcja anonimowa w zapisie znanym z rachunku lambda (w drugiej wersji dwukrotnie; patrz także Podrozdział 5.3). Zwrócić trzeba tu uwagę na to, że funkcje `silnia1`, `silnia2` zdefiniowane są za pomocą zwykłego podstawienia — zapis taki (z podstawieniem i słowem `lambda` jest równoważny odpowiedniej definicji z użyciem słów `def` oraz `return` — z tą tylko uwagą, że zawartość instrukcji `lambda` w Pythonie ogranicza się do pojedynczego wyrażenia (co jednak w programowaniu funkcyjnym zwykle wystarcza).

— Wiersze 29–35 prezentują trzy równoważne (ale napisane różnymi technikami) definicje funkcji składającej dwie funkcje — to jest: wynikiem każdej z funkcji `zlozenie1`, `zlozenie2`, `zlozenie3` jest funkcja będąca złożeniem jej argumentów.

— Wiersze 37–43 pokazują zastosowanie właśnie zdefiniowanej funkcji do wygenerowania (i użycia w praktyce) funkcji obliczającej sinus kąta podanego w stopniach. W wierszach 37 i 42 można użyć dowolnej z funkcji `zlozenie1`–`3`.

— W końcu funkcja `samozlozenie` zwraca funkcję będącą jej argumentem złożonym z samym sobą `n` razy. I jej użycie — Czytelnik sam na pewno już się orientuje, co tu się dzieje.

— Wiele podobnych funkcji jest zdefiniowanych w Pythonie standardowo jako funkcje wbudowane (na przykład `map`, `filter`, `reduce`), w modu-

[27] Jeśli chodzi o efektywność tych funkcji, to na pewno pozostawiają wiele do życzenia, ale tu chodzi tylko o przykład (nieco sztuczny, to prawda).

le `operator` (na przykład `mul`, `add`), w module `functools` (na przykład `partial`); a także w zewnętrznym module `functional` [80].

7.6.2. Leniwa ewaluacja i sekwencje nieskończone

Duża siła programowania funkcyjnego tkwi w leniwych obliczeniach. Python jest z natury (jak większość imperatywnych języków) językiem gorliwym, jednakże ma narzędzia do programowania obliczeń leniwych. Pierwsze z nich to *iteratory*. Iterator jest obiektem reprezentującym strumień danych, ale — w przeciwieństwie do list czy krotek — nie udostępnia tych danych za jednym zamachem, lecz w miarę potrzeb, zwykle po jednej. Z drugiej strony, taki obiekt iteratorowy może być użyty w wielu miejscach, w których potrzeba sekwencji (na przykład po `in` w warunku lub w pętli `for`).

Zdecydowaną przewagę iteratorów widać w sytuacji, gdy mamy do przejrzenia bardzo długą sekwencję (może nawet potencjalnie nieskończoną!), ale jest duża szansa, że to przeglądanie może się skończyć wcześnie (schemat na Listingu 7.46). Jeśli jako sekwencji użyjemy listy, to będzie ona musiała być utworzona w pamięci w całości (ze względu na gorliwość Pythona), musi być więc skończona, a co więcej mieścić się w ograniczonej pamięci. Oprócz pamięci, samo utworzenie listy zajmie znaczny czas, jeśli lista jest duża. Jeśli zaś w roli sekwencji użyjemy iteratora, to nie ma takiego problemu — dane będą tworzone w pamięci na bieżąco, gdy są potrzebne, a dane użyte i już niepotrzebne mogą być z pamięci wyrzucone w procesie zbierania nieużytków.

Listing 7.46. Schemat częściowego przeglądania sekwencji

```
for el in sekwencja:
    zrob_cos(el)
    if mozna_skonczyc(el):
        break
```

Żeby obiekt mógł działać jako iterator musi mieć odpowiedni protokół (bo, jak wiemy, w Pythonie stosowane jest kacze typowanie), nic więcej. Musi więc mieć metodę bezargumentową[28] `next`, która zwraca jedną daną — o ile jest co zwracać — albo wyrzuca wyjątek `StopIteration` oraz metodę tożsamościową o sygnaturze `__iter__(self)` (służy ona tylko do zawiadomienia Pythona, że definiowana jest klasa iteratorowa). Listing 7.47 pokazuje dwa iteratory, z których pierwszy jest skończony, a drugi nie.

Listing 7.47. Iteratory jako klasy

[28] Właściwie — jak wszystkie metody w Pythonie — musi ona mieć jeden argument, obowiązkowy dla metod: `self`.

```
class Powtarzacz(object):
    def __init__(self, ile, czego):
        self.licznik = ile
        self.co = czego
    def next(self):
        if self.licznik > 0:
            self.licznik -= 1
            return self.co
        raise StopIteration
    def __iter__(self):
        return self

class Zapetlacz(object):
    def __init__(self, *a):
        self.licznik = -1
        self.dane = a
    def next(self):
        self.licznik += 1
        if self.licznik >= len(self.dane):
            self.licznik = 0
        return self.dane[self.licznik]
    def __iter__(self):
        return self

it1 = Powtarzacz(10, 1)
it2 = Powtarzacz(5, 'ala')
it3 = Zapetlacz(1,2,3,4,5,6)

print it1, it2, it3

print list(it1)
for el in it2:
    print el
#print list(it3)
# wyzej zakomentowane, bo by sie zawiesilo
for el in it3:    # a to sie nigdy nie konczy
    print el
```

Python dysponuje wieloma standardowymi narzędziami, które zwracają iteratory. Przede wszystkim są to wbudowane `xrange` (działające jak `range`, ale zwracające iterator zamiast listy) oraz `enumerate` (patrz strona 150 i dalej). Dalej mamy do dyspozycji cały moduł `itertools`, z którego warto wymienić następujące (ale nie wszystkie!) iteratory:

— `count(n)` (domyślnie `n=0`) daje `n`, `n+1`, `n+2`... — bez końca;
— `cycle(sekwencja)` daje elementy `sekwencji`, a jak się skończą, zaczyna od nowa;
— `repeat(dana, n)` (domyślnie $n=\infty$) daje `n` razy powtórzoną `daną`;
— `izip(sekw1, ...)` daje sekwencję krotek złożonych z odpowiadających

sobie elementów danych sekwencji (odpowiednik standardowej funkcji `zip` dla list);
— `imap(funkcja, sekw1, ...)` daje sekwencję wyników działania funkcji na argumentach pochodzących po jednym z każdej danej sekwencji (odpowiednik standardowej funkcji `map` dla list);
— `ifilter(predykat, sekwencja)` daje sekwencję tych elementów danej `sekwencji`, dla których `predykat`[29] jest spełniony;
— `takewhile(predykat, sekwencja)` daje z powrotem `sekwencję`, ale uciętą przed pierwszym elementem niespełniającym `predykatu`;
— `dropwhile(predykat, sekwencja)` daje z powrotem `sekwencję`, ale odrzucając jej początkowe (i tylko początkowe) elementy spełniające `predykat`.

7.6.3. Generatory

Jeśli zwracane wartości mają być bardziej skomplikowane, używanie iteratorów staje się nieco utrudnione. A to ze względu na fakt, że obiekt-iterator musi przechowywać swój stan, co może czasem być organizacyjnie kłopotliwe i sprzyjające pomyłkom i przeoczeniom. Python jednak udostępnia kolejne narzędzie — *generatory*.

Generatory są drugim rodzajem podprogramów (po funkcjach). Definiuje się je dokładnie tak samo, jak zwykłe funkcje, ale żeby był to generator, wartości muszą być zwracane za pomocą instrukcji `yield` zamiast `return`[30].

Jak działa generator? Otóż, wywołanie generatora tworzy obiekt iteratorowy. Iterator ten za każdym razem, gdy jest proszony o dane (metoda `next`) wykonuje po prostu ciało funkcji do napotkania instrukcji `yield`. Jednakże wykonanie instrukcji `yield`, zwracając wynik (jak `return` w zwykłej funkcji), nie przerywa wykonywania podprogramu, lecz je **zawiesza** — do momentu zapotrzebowania na następną daną, kiedy to działanie treści generatora jest wznawiane **od chwili ostatniego zawieszenia** w tym samym, zapamiętanym, stanie[31]. Dopiero wykonanie instrukcji `return` (lub dojście sterowania do końca treści generatora) kończy sekwencję.

Listing 7.48 pokazuje to samo, co Listing 7.47, lecz z użyciem generatorów.

Listing 7.48. Generatory

```
1 def genPowtarzacz(ile, czego):
      for i in range(ile):
```

[29] Predykat to funkcja dająca wartości logiczne — jak w programowaniu logicznym.
[30] Instrukcji `return` można użyć w generatorze, ale tylko bez argumentów, jako instrukcji zakończenia podprogramu.
[31] Tę właściwość można też wykorzystać do implementacji w Pythonie *współprocedur* za pomocą generatorów [33]

```
        yield czego

def genZapetlacz(*a):
    while True:
        for el in a:
            yield el

it1 = genPowtarzacz(10, 1)
it2 = genPowtarzacz(5, 'ala')
it3 = genZapetlacz(1,2,3,4,5,6)

print it1, it2, it3

print list(it1)
for el in it2:
    print el
#print list(it3)
# wyzej zakomentowane, bo by sie zawiesilo
for el in it3:    # a to sie nigdy nie konczy
    print el
```

7.6.4. Listy składane; wyrażenia generatorowe

Python zapożyczył z języków funkcyjnych (konkretnie: z Haskella, patrz Podrozdział 5.6.1.1 na stronie 97) jeszcze jeden, typowy dla nich, lukier składniowy: *listy składane* (ang. *list comprehension*) oraz *wyrażenia generatorowe*. Ogólna składnia listy składanej ma formę pokazaną na Listingu 7.49.

Listing 7.49. Ogólna postać listy składanej w Pythonie

```
[ wyrazenie   for z1 in sekw1   if war1
              for z2 in sekw2   if war2
              ...
              for zn in sekwn   if warn ]
```

W liście składanej każda z fraz `if warx` może zostać opuszczona (i jest wtedy traktowana jak `if True`). Jest to mniej więcej równoważne kodowi z Listingu 7.50 — oczywiście normalnie nie jest to wszystko podstawiane do jakiejkolwiek zmiennej (a w Listingu 7.50 jest to zmienna `wynik`).

Listing 7.50. Imperatywny odpowiednik listy składanej

```
wynik = []
for z1 in sekw1:
  if war1:
    for z2 in sekw2:
      if war2:
        ...
```

```
            for zn in sekwn:
8             if warn:
                wynik.append(wyrazenie)
```

Kilka przykładów list składanych pokazuje Listing 7.51. Pozostawiamy Czytelnikowi sprawdzenie działania tych wyrażeń w praktyce oraz ich analizę.

Listing 7.51. Przykłady list składanych

```
1 [(x, d) for x in range(1, 11)
            for d in range(1, 11) if x%d == 0]
3 [2**n for n in range(21)]
  [s.upper()
5   for s in 'Ala ma kota i trzy wielkie psy.'.split()]
  [w for w in open('deskryptory.py', 'r')
7       if 'def ' in w or 'class ' in w]
```

Zamiast list składanych często warto używać wyrażeń generatorowych. Podobnie jak iteratory mogą mieć pewne przewagi nad listami (ze względu na lenistwo, strona 174, Podrozdział 7.6.2), tak też wyrażenia generatorowe nad listami składanymi.

Wyrażenie generatorowe składniowo wygląda dokładnie tak, jak lista składana, ale zewnętrzne nawiasy nie są kwadratowe `[]`, lecz okrągłe `()`. Semantyka takiego wyrażenia jest całkiem podobna do listy składanej (a więc wartość i kolejność elementów wyniku jest taka sama), jednakże bezpośrednim wynikiem wyrażenia generatorowego nie jest lista, tylko **iterator**, który kolejne elementy dostarcza w miarę potrzeb.

7.6.5. Dekoratory

Dekorator jest funkcją, która opakowuje inną funkcję, zwracając jej zmodyfikowaną wersję. Rozważmy sytuację, w której pisząc jakieś oprogramowanie chcemy śledzić (na przykład w celu odpluskwiania) wywołania pewnych funkcji (ale chcemy też łatwo włączać i wyłączać to śledzenie). Listing 7.52 pokazuje podejście wykorzystujące wiedzę dotychczas poznaną, natomiast Listing 7.53 pokazuje to samo z notacją dekoratorową (która jest tylko lukrem składniowym). Łatwo teraz — dzięki dekoratorom — wyłączać i włączać śledzenie poszczególnych funkcji (wystarczy znak komentarza `#` w odpowiednim miejscu).

Listing 7.52. Dekoratory bez notacji dekoratorowej

```
1 def sledz(f):
      def sledzenie(*a, **k):
```

```
        print 'START FUNKCJI: %s(*%s, **%s)' % (
            f.__name__, repr(a), repr(k))
        wynik = f(*a, **k)
        print 'KONIEC FUNKCJI: %s(*%s, **%s) ---> %s' % (
            f.__name__, repr(a), repr(k), repr(wynik))
        return wynik
    sledzenie.__name__ = f.__name__
      #zeby funkcja zachowala nazwe po udekorowaniu
    return sledzenie

def suma(x, y):
    return x+y
#suma = sledz(suma)

def iloczyn(x, y):
    return x*y
iloczyn = sledz(iloczyn)

def przeciw(x):
    return -x
przeciw = sledz(przeciw)

print suma(iloczyn(4, 5),
           suma(4, iloczyn(przeciw(10), 2)))
print suma(suma('Ala', ' ma '),
           suma('kota', iloczyn('!', 3)))
```

Listing 7.53. Dekoratory z notacją dekoratorową

```
def sledz(f):
    def sledzenie(*a, **k):
        print 'START FUNKCJI: %s(*%s, **%s)' % (
            f.__name__, repr(a), repr(k))
        wynik = f(*a, **k)
        print 'KONIEC FUNKCJI: %s(*%s, **%s) ---> %s' % (
            f.__name__, repr(a), repr(k), repr(wynik))
        return wynik
    sledzenie.__name__ = f.__name__
      #zeby funkcja zachowala nazwe po udekorowaniu
    return sledzenie

#@sledz
def suma(x, y):
    return x+y

@sledz
def iloczyn(x, y):
    return x*y

@sledz
```

```
22 def przeciw(x):
       return -x
24
   print suma(iloczyn(4, 5),
26             suma(4, iloczyn(przeciw(10), 2)))
   print suma(suma('Ala', ' ma '),
28             suma('kota', iloczyn('!', 3)))
```

7.7. Pytania i zadania

1. Wyjaśnij pojęcia: kacze typowanie, EAFP, LBYL, zmienialność i niezmienialność, parametry pozycyjne, parametry nazwane, inicjalizator, metoda specjalna, refleksyjność, deskryptor, domknięcie, iterator, generator, lista składana, wyrażenie generatorowe, dekorator.
2. Czym różni się lista od krotki w Pythonie? Czym różnią się od podobnych pojęć w Haskellu?
3. Dlaczego przy konstrukcji jednoelementowej krotki musimy dać „nadmiarowy" przecinek: `(4,)`? Dlaczego po prostu `(4)` nie może oznaczać krotki?
4. Dlaczego (dla dowolnej ustalonej i istniejącej zmiennej `x`) `id(x) == id(x)` daje zawsze `True`, natomiast `id(x) is id(x)` zwykle daje `False`?
5. Napisz w Pythonie program, który sprawdzi, jaki jest zakres liczb całkowitych generowanych przed rozpoczęciem wykonywania programu i przechowywanych potem na wszelki wypadek.
6. Rozważmy inną próbę poprawienia Listingu 7.21 ze strony 150 — przedstawia ją poniższy Listing 7.54.

Listing 7.54. Szukanie pozycji cyfr — krotki zamiast list

```
   napis = raw_input("Podaj napis zlozony z samych cyfr: ")
2  wynik = 10*[()]
   for (i, z) in enumerate(napis):
4    wynik[int(z)] += (i,)
   print wynik
```

Zmiany dotyczą tylko linii 2 (pusta krotka zamiast pustej listy) oraz 4 (do krotki musimy dodawać krotkę, nie listę), poza tym algorytm nie zmienił się. Wyjaśnij: (a) dlaczego teraz program działa poprawnie (poza tym, że dostajemy listę krotek, a nie listę list); (b) dlaczego jest to rozwiązanie znacznie mniej efektywne (szczególnie dla dużych danych wejściowych) od kodu 7.22; (c) dlaczego owa pogorszona efektywność nie stoi w sprzeczności z podanym na stronie 144 (Podrozdział 7.3.1.3) zdaniem, że krotki są nieco szybsze od list (innymi słowy: jakie dodatkowe

czynności są tu wykonywane ponad te, które wynikają z Listingów 7.21 oraz 7.22).

7. Napisz w Pythonie funkcję `moje_range`, która przyjmuje od jednego do trzech parametrów i naśladuje działanie standardowej funkcji `range`, ale działa dla argumentów zmiennoprzecinkowych. Ma więc zachodzić:

```
moje_range(n) == moje_range(0.0, n)
moje_range(a, b) == moje_range(a, b, 1.0)
moje_range(a, b, k) == [a, a+k, ..., x]
```

   przy czym dla `k > 0.0` ma być `x < b <= x+k`, a dla `k < 0.0` ma być `x > b >= x+k`; jeśli takie `x` nie istnieje, to wynikiem ma być `[]`. Dla `k == 0.0` zgłaszamy wyjątek `ValueError`.
8. Przerób funkcję z poprzedniego zadania na iterator oraz na generator.
9. Do implementacji liczb wymiernych z Listingu 7.37 dopisz pozostałe potrzebne operacje:
   — arytmetyczne dwuargumentowe: `__sub__`, `__mul__`, `__div__`, `__pow__`, `__rsub__`, `__rmul__`, `__rdiv__`, `__rpow__`;
   — arytmetyczne dwuargumentowe: `__neg__` (zmiana znaku), `__pos__` (zachowanie znaku), `__abs__` (wartość bezwzględna);
   — porównania: `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__` (zamiast tych sześciu metod można zdefiniować jedną: `__cmp__`);
   — automatyczne konwersje: `__int__`, `__long__`, `__float__` (te trzy już właściwie są, ale pod innymi nazwami), `__complex__`, `__nonzero__` (ta ostatnia to konwersja na `bool`).
10. * Napisz klasę `ListaDom` potomną dla klasy `list`, która w przypadku chęci pobrania nieistniejącego elementu (na przykład `print lista[10]` dla listy trójelementowej) zwracała daną podaną jako domyślna przy inicjalizacji. Wskazówki: przedefiniuj metodę specjalną `__getitem__`; nie wpadnij w nieskończoną rekurencję, lecz użyj `list.__getitem__(self, indeks)` (analogicznie do Listingu 7.39 na stronie 167).
11. * Napisz klasę `Wektor`, która będzie implementować różne operacje na wektorach w sensie matematycznym (takie jak: zwracanie składowej wektora, zwracanie wymiaru wektora, dodawanie, odejmowanie, mnożenie przez liczbę, iloczyn skalarny, norma, kopiowanie wektora, zmiana elementów). Zadbaj o kontrolę zgodności wymiarów wektorów w operacjach, które tego wymagają (na przykład dodawanie wektorów).
12. * Napisz klasę `Macierz` z odpowiednimi operacjami i zintegruj ją z klasą `Wektor` z poprzedniego zadania.
13. * Napisz klasę deskryptorową, która kontroluje typ podstawianej do

deskryptora danej i wyrzuca wyjątek, gdy nie jest zgodny z typem podanym przy inicjalizacji atrybutu.

14. * Napisz iterator oraz generator, które wytwarzają nieskończoną sekwencję liczb losowych z pewnego ustalonego — w momencie inicjalizacji — przedziału.
15. Zapisz iteratory `it1`, `it2`, `it3` z Listingu 7.47 jedynie przy użyciu standardowych narzędzi modułu `itertools`.
16. * Rozważmy następujące wyrażenie w Pythonie: `"ala"+" "+"ma"+" "+"kota"` — oraz analogiczne w Haskellu: `"ala"++" "++"ma"++" "++"kota"`. W Pythonie, przez wzgląd na nieefektywność, jesteśmy zniechęcani do używania takich wyrażeń (lepiej napisać na przykład `"".join(["ala", " ", "ma", " ", "kota"])` albo `"%s %s %s" % ("ala", "ma", "kota")`). W Haskellu natomiast nie ma takiego problemu. Dlaczego?
17. * Spróbuj napisać iteratory lub generatory, działające na wzór definicji z zadania 11 ze strony 110.
18. * Spróbuj zastosować schemat pracy z sekwencją nieskończoną definiując w Pythonie szukanie przybliżenia $\sqrt{a}$ metodą Newtona-Raphsona na wzór rozwiązania z Podrozdziału 5.6.2.2 ze strony 100.
19. * Zdefiniuj w Pythonie klasę (wraz z metodami) definiującą binarne drzewa poszukiwań (patrz też strona 98, Podrozdział 5.11).

Rozdział 8

Programowanie niesekwencyjne

8.1. Programowanie współbieżne, równoległe, rozproszone

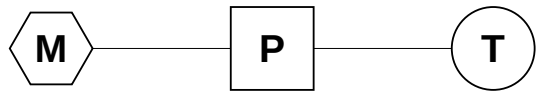
Do tej pory omawialiśmy programowanie sekwencyjne. W programowaniu niesekwencyjnym (współbieżnym, równoległym, rozproszonym) trzeba zmierzyć się z całkiem nowymi problemami, niespotykanymi w programowaniu sekwencyjnym. Są to między innymi:
— niedeterminizm,
— komunikacja,
— synchronizacja,
— podział i rozsyłanie danych,
— zrównoważanie obciążenia,
— odporność na niepowodzenia,
— heterogeniczność,
— pamięć wspólna i rozproszona,
— zakleszczenia,
— wzajemne wykluczanie,
— rywalizacja,
— głodzenie,
i wiele, wiele innych...

Problem podzielić należy na części, które mogą być wykonywane równolegle tzn. które nie muszą być wykonywane sekwencyjnie (jedna po drugiej). Paradygmat programowania równoległego [37, 56, 77] jest to paradygmat rozdzielający proces wykonania programu na wiele procesorów lub wiele maszyn w jednym czasie. W idealnym przypadku n procesorów miałoby moc przetwarzania n-razy większą od pojedynczego procesora. Każdy z procesorów wykonywałby $1/n$ część obliczeń. Niestety taki przyrost mocy jest nieosiągalny z trzech głównych powodów:
— trudność w podzieleniu problemu na n równych części dających się przetwarzać równolegle (istnieją części z natury sekwencyjne);
— konieczność przesyłania danych między procesami (narzut na komunikację);
— konieczność synchronizacji pomiędzy procesami (czekanie jeden na drugiego).

Programowanie równoległe wykorzystuje języki zorientowane obiektowo lub proceduralnie, dodając do nich mechanizm synchronizacji procesów i wątków oraz współdzielenie pamięci i innych zasobów komputera. Synchronizacja procesów (ang. *process synchronization*) to zabieg mający na celu ustalenie właściwej kolejności działania procesów współpracujących, w szczególności procesów korzystających ze zmiennych dzielonych.

Równoległość oznacza równoczesne wykonanie zadań przez wiele procesorów. O programowaniu rozproszonym mówimy, gdy mamy wiele procesorów połączonych siecią (ale nie wieloprocesorowy komputer). Z drugiej

strony pokrewne jest równoległemu programowanie współbieżne, w którym możemy mieć tylko jedną jednostkę przetwarzającą, ale wiele zadań wykonywanych przy pomocy dzielenia czasu procesora. Rysunki 8.1–8.4 pokazują te trzy modele programowania (oraz programowanie sekwencyjne); **M** oznacza tu pamięć (ang. *memory*), **P** — procesor, **T** — zadanie (ang. *task*); natomiast linie łączące poszczególne figury oznaczają komunikację i przynależność do siebie danych elementów.



Rysunek 8.1. Model działania algorytmu sekwencyjnego

Modele programowania równoległego zależą od architektury komputerów i od komunikacji między zadaniami.

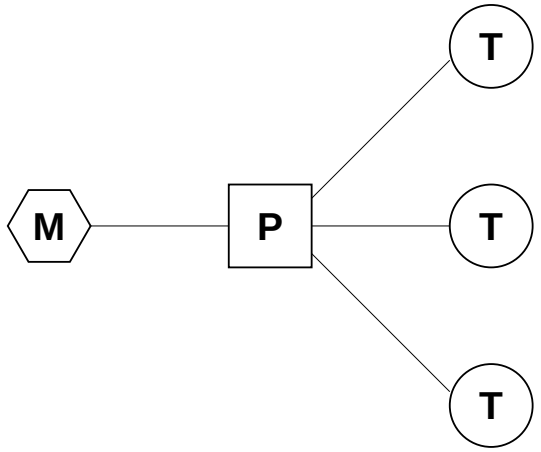
Model programowania z pamięcią wspólną (dzieloną) — czyli z równoległością danych. Porozumiewanie się następuje przez zmianę wartości umówionych komórek pamięci. Model ten jest łatwy do stosowania dla komputerów o pamięci wspólnej, na przykład wieloprocesorowych.

Model programowania z pamięcią rozproszoną — czyli z przesyłaniem komunikatów. Metoda ta polega na przesyłaniu wiadomości za pomocą wywołania odpowiednich podprogramów. Metoda ta wymaga dużego wsparcia ze strony systemu operacyjnego. Stosowana jest do architektury komputerów z pamięcią rozproszoną.

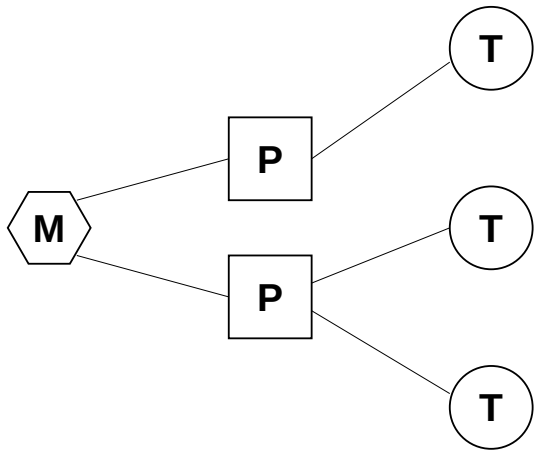
Do prowadzenia obliczeń równoległych oprócz sprzętu o architekturze wspomagającej operacje równoległe (procesory wielordzeniowe, symetryczne wieloprocesorowe, systemy składające się z wielu maszyn: klastry, systemy MPP, gridy) potrzeba równoległych algorytmów.

Pierwszym krokiem w zrównoleglaniu algorytmu jest podział problemu na zadania (ang. *task*), które mogą być wykonane niezależnie. Podział ten nazywany jest dekompozycją. Problem może być podzielony na zadania na wiele różnych sposobów. Zadania mogą być takich samych bądź też różnych rozmiarów. Podział zbioru na zadania może być zilustrowany w postaci grafu skierowanego, którego wierzchołki odpowiadają zadaniom, a krawędzie wskazują, że wynik jednego z zadań jest potrzebny drugiemu. Graf ten nazywany jest grafem zależności zadań (ang. *task dependency graph*). Liczba zadań na które dzielimy problem określają nam ziarnistość dekompozycji:

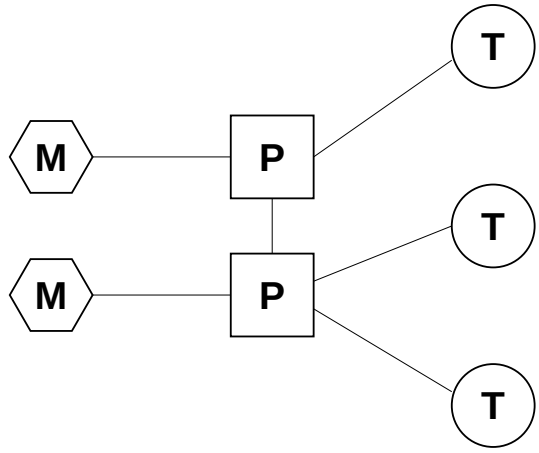
— *dekompozycja gruboziarnista* (ang. *coarse grained*) — problem podzielony na niewielką liczbę stosunkowo dużych zadań;
— *dekompozycja drobnoziarnista* (ang. *fine grained*) — problem podzielony na dużą liczbę niewielkich zadań.



Rysunek 8.2. Model działania algorytmu współbieżnego nierównoległego



Rysunek 8.3. Model działania algorytmu równoległego nierozproszonego (z pamięcią wspólną)



Rysunek 8.4. Model działania algorytmu rozproszonego

Stopień współbieżności (ang. *degree of concurrency*) — jest liczbą zadań, które mogą być wykonywane równolegle. Może się on zmieniać w trakcie porażki wykonywania programu, stąd wyróżniamy maksymalny stopień współbieżności oraz średni stopień współbieżności. Stopień współbieżności zwiększa się, gdy dekompozycja staje się bardziej drobnoziarnista i na odwrót. Nie oznacza to, że dekompozycja drobnoziarnista prowadzi do lepszej wydajności. Zwiększanie ziarnistości prowadzi bowiem do większej interakcji pomiędzy zadaniami, co zwiększa czas wykonania programu (koszty komunikacji).

Możemy wliczyć wiele technik dekompozycji danych.

Dekompozycja rekurencyjna. Jest dobra dla problemów, które mogą być rozwiązane przy pomocy paradygmatu „dziel i zwyciężaj" (ang. *divide and conquer*). Problem jest dekomponowany na podproblemy o mniejszym wymiarze. Podproblemy są niezależne od siebie i mogą być rozwiązane równolegle. Rozwiązanie problemu wykorzystuje rozwiązania podproblemów. Podproblemy są rekurencyjnie dekomponowane dalej, aż do osiągnięcia wymaganej ziarnistości. Po zakończeniu dekompozycji podproblemy stają się zadaniami.

Dekompozycja danych. Jest to technika powszechnie używana do problemów operujących na dużych ilościach danych. Dekompozycja wykonywana jest w dwóch krokach. W kroku pierwszym dzielone są dane, na których przeprowadzane są obliczenia. W kroku drugim ten podział danych wykorzystywany jest do podziału obliczeń. Podział danych może być wykorzystany na wiele sposobów;

1. podział danych wyjściowych (wyników) — jest możliwy do wykona-

nia jeżeli element wyników może być obliczony niezależnie od pozostałych jako funkcja danych wejściowych;
2. podział danych wejściowych;
3. podział danych pośrednich;
4. dowolna kombinacja powyższych trzech metod.

Dekompozycja eksploracyjna. Dekompozycja eksploracyjna jest wykorzystywana w przypadku problemów, których rozwiązanie wymaga przeszukania pewnej przestrzeni rozwiązań (ang. *solution space*). W dekompozycji eksploracyjnej dzielimy przestrzeń rozwiązań na części i przeszukujemy niezależnie każdą z tych części do momentu znalezienia rozwiązania. Dekompozycja eksploracyjna różni się od rekurencyjnej, tym że niezakończone zadania są kończone, gdy tylko zostanie znalezione rozwiązanie.

Dekompozycja spekulacyjna. Dotyczy realizacji zbioru możliwych zadań bez znanych wartości danych wejściowych. Zadania te są przetwarzane w możliwym stopniu, zazwyczaj tylko jedno z zadań będzie kontynuowane jako wynik uzyskania wartości danych wejściowych decydujących o ścieżce obliczeń.

Dekompozycja hybrydowa. Jest połączeniem niektórych spośród czterech poprzednich.

Metodyka konstruowania algorytmów równoległych podana przez Iana Fostera [15] to:
1. podział na podzadania;
2. określenie komunikacji: lokalna/globalna, strukturalna/niestrukturalna, statyczna/dynamiczna, synchroniczna/niesynchroniczna;
3. grupowanie zadań — zadania są grupowane w większe zadania dla poprawy wydajności i redukcji kosztów komunikacji globalnej (bo lokalna jest zwykle dużo tańsza);
4. przypisanie zadań konkretnym procesorom.

8.2. Model programowania z pamięcią wspólną

Programowanie multiprocesorów musi uwzględniać ich architekturę, w szczególności sposób dostępu do pamięci. Procesory mogą mieć równy lub różny dostęp do tego samego obszaru pamięci.

Podstawowe funkcjonalności w modelu programowania z pamięcią dzieloną to:
— narzędzia tworzenia i kończenia wątków lub procesów;
— synchronizacja dostępu do obszarów pamięci dzielonej.
Przykładem wieloplatformowego interfejsu programowania aplikacji równo-

ległych dla komputerów o architekturze wieloprocesorowej z pamięcią dzieloną jest OpenMP (ang. *Open Multi-Processing*).

8.2.1. OpenMP

OpenMP [45] to standard definiujący zbiór dyrektyw kompilacji, funkcji bibliotecznych i zmiennych środowiskowych mających pomóc w tworzeniu programów równoległych w systemach z pamięcią dzieloną. Atutem standardu jest fakt, iż większość producentów sprzętu równoległego wspiera OpenMP. OpenMP rozszerza sekwencyjny model programowania o *Single Program Multiple Data* (SPMD), podział pracy (*work-sharing*) i synchronizację oraz wspomaga operowanie na wspólnych i prywatnych danych. Równoległość programu programista musimy wskazać jawnie i do niego należy przewidywanie wszelkich zależności, konfliktów i uwarunkowań.

Cechy standardu OpenMP:
— przenośny (wspierany przez czołowych producentów sprzętu komputerowego);
— skalowalny;
— elastyczny;
— przyjazny dla programisty.

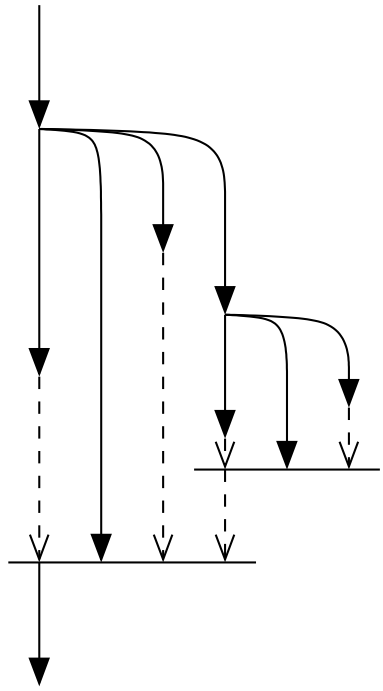
Program zaczyna działanie jako pojedynczy wątek, wątek główny (*master thread*), aż do momentu napotkania konstrukcji równoległej. W tym momencie utworzona zostaje grupa wątków, przy czym wątek główny staje się nadrzędny w stosunku do pozostałych. Dalej każdy wątek wykonuje program znajdujący się w dynamicznym rozszerzeniu konstrukcji równoległej (model SPMD) poza obszarami, w których program jest wykonywany w modelu pracy dzielonej. Po zakończeniu pracy w konstrukcji równoległej wątki zostają zsynchronizowane niejawną barierą i tylko wątek główny kontynuuje pracę. Taki model pracy nazywany jest *fork-join* (Rysunek 8.5).

W programie można użyć dowolnej liczby konstrukcji równoległych. Można również używać dyrektyw w funkcjach, które są wywoływane z konstrukcji równoległych.

Dyrektywy standardu OpenMP bazują na dyrektywie `#pragma` zdefiniowanej w standardach C i C++. Ogólny format dyrektyw jest następujący:

```
#pragma omp dyrektywa [klauzula [klauzula] ... ]
```

przy czym każda dyrektywa rozpoczyna się i kończy wraz z początkiem i końcem wiersza programu. Dyrektywa odnosi się do występującego bezpośrednio po niej bloku strukturalnego. Każda dyrektywa posiada swój własny zestaw dopuszczalnych klauzul. Zasadniczo porządek klauzul nie ma znaczenia i mogą one być zagnieżdżone.



Rysunek 8.5. Przykładowa praca wątków w modelu fork-join (strzałki ciągłe to działanie wątków, strzałki przerywane to ich bezczynność, linie poziome to bariery)

Konstrukcja *parallel*. Dyrektywa `parallel` definiuje obszar równoległy, wykonywany przez grupę wątków. Jest to podstawowa konstrukcja rozpoczynająca pracę równoległą.

```
#pragma omp parallel [klauzula [klauzula] ... ]
{
   ...
}
```

Klauzulą może być:

— `if (warunek)` — uzależnienie wykonania równoległego od spełnienia warunku;
— `private (lista zmiennych)` — każda zmienna z listy będzie odrębną zmienną lokalną w każdym wątku;
— `firstprivate (lista zmiennych)` — jak wyżej, ale zmienne są dodatkowo inicjowane na początku sekcji;
— `default (shared)` — wszystkie zmienne są domyślnie wspólne;
— `default (none)` — żadna zmienna nie jest domyślnie wspólna;

— `shared (lista zmiennych)` — wskazanie zmiennych wspólnych dla wszystkich wątków w danym bloku równoległym;
— `reduction (operator, lista zmiennych)` — przeprowadzenie redukcji za pomocą operatora za pomocą listy zmiennych.

Kiedy wątek napotyka konstrukcję `parallel`, wtedy zostaje stworzona grupa wątków pod warunkiem, że w dyrektywie nie występuje klauzula `if`, lub też że wyrażenie skalarne w `if` sprowadza się do niezerowej wartości. Wówczas wątek tworzący grupę staje się wątkiem głównym grupy i wszystkie wątki w grupie, włącznie z głównym, wykonują obszar programu współbieżnie. Liczba wątków w grupie jest kontrolowana przez zmienne środowiskowe lub wywołania funkcji bibliotecznych.

Gdy wartość wyrażenia skalarnego w `if` jest zerowa, to obszar jest wykonywany sekwencyjnie.

Na końcu obszaru równoległego znajduje się domyślna bariera. Po opuszczeniu obszaru równoległego pracę kontynuuje jedynie wątek główny grupy. Jeśli któryś z wątków wykonujących obszar równoległy napotka następną konstrukcję `parallel`, wówczas tworzy nową grupę i staje się jej wątkiem głównym.

Domyślnie zagnieżdżone konstrukcje równoległe są usekwencyjniane, co w rezultacie oznacza, że grupa składa się z jednego wątku. Zachowanie to może zostać zmienione za pomocą funkcji `omp_set_nested` lub zmiennej środowiskowej `OMP_NESTED`.

Konstrukcja *for*. Konstrukcja `for` identyfikuje pętlę `for`, która zostanie wykonana równolegle. Iteracje pętli zostaną rozdzielone pomiędzy istniejące wątki. Składnia wygląda następująco:

```
#pragma omp for [klauzula [klauzula] ... ]
for(...) ...
```

Klauzulą może być:
— `private (lista zmiennych)`, `firtsprivate (lista zmiennych)`, `reduction (operator, lista zmiennych)` — jak w `parallel`;
— `lastprivate (lista zmiennych)` — wskazuje listę zmiennych lokalnych, których wartość na końcu byłaby taka jak przy wykonywaniu sekwencyjnym;
— `schedule (rodzaj [, porcja])` — określenie sposobu podziału iteracji między wątki;
— `nowait` — pozostałe wątki nie oczekują na zakończenie (bez tej klauzuli, na końcu jest niejawna bariera).

Klauzula `schedule` określa sposób w jaki iteracje zostaną rozdzielone na wątki z grupy. Poprawność programu nie powinna zależeć od tego, który z

wątków wykona daną iterację. Wartość `porcja`, o ile wystąpiła w klauzuli, powinna być niezależnym od wykonywanej pętli wyrażeniem całkowitym większym od zera. Jeżeli nie została użyta klauzula `nowait`, to na końcu konstrukcji `for` znajduje się domyślna bariera.

Listing 8.1. Pierwszy przykład użycia pragm OpenMP

```
int KopiujMacierz(int n,double z[N][N],double pomz[N][N]) {
  //Funkcja ma na celu skopiowanie macierzy
  int i, k;
#pragma omp parallel default(shared)
  {
#pragma omp for shared(pomz, z, n)  private(i, k)
    for (i=1; i<=n; i++) {
      for (k=1; k<=n; k++) {
        pomz[i][k]= z[i][k];
      } //end for k
    } //end for i
  } //end pragma parallel
  return 0;
} //end KopiujMacierz
```

Funkcja z Listingu 8.1 kopiuje macierz `z` do macierzy `pomz`. Programista wykorzystuje dyrektywę `parallel`, aby wskazać kompilatorowi obszar kodu, który będzie zrównoleglany.

Kolejna dyrektywa — `for` — informuje kompilator, że zrównoleglana będzie pętla typu `for`. Następnie określa się, które zmienne, będą zmiennymi wspólnymi (`shared`), a które prywatnymi (`private`). Zmienne wspólne są dostępne dla każdego wątku, natomiast każdy wątek ma swój zestaw zmiennych prywatnych. W funkcji `KopiujMacierz` zmiennymi prywatnymi są `i`, `k`, czyli liczniki pętli. Uczynienie ich zmiennymi prywatnymi powoduje, że każdy wątek ma swój wewnętrzny licznik iteracji. Każdy wątek otrzymuje kopie zmiennych `k` oraz `i`, i na tych kopiach pracuje. Zmiennymi wspólnymi są takie zmienne, których wartości w trakcie obliczeń nie zmieniają się, ale za pomocą których te obliczenia wykonujemy. W programie zmiennymi wspólnymi są: macierz `z` i macierz `pomz` oraz zmienna określająca rozmiar macierzy kwadratowej `n`.

Nie podano liczby wątków, które program ma wykorzystać. OpenMP wykrywa liczbę dostępnych wątków i przydziela każdemu odpowiednią liczbę iteracji do wykonania, chociaż programowo można to zmienić.

Konstrukcja *sections*. Dyrektywa `sections` identyfikuje zbiór sekcji, które zostaną rozdzielone do wykonania wątkom w grupie. Każda sekcja zostanie wykonana dokładnie raz przez jeden z wątków z grupy. Składnia jest następująca:

```
#pragma omp sections [klauzula [klauzula] ...]
{
[#pragma omp section]
        { ... }
[#pragma omp section]
        { ... }
...
}
```

Klauzule dopuszczalne tutaj to znane z poprzednich konstrukcji: `private (lista_zmiennych)`, `firstprivate (lista_zmiennych)`, `lastprivate (lista_zmiennych)`, `reduction (operator, lista zmiennych)`, `nowait`.

Funkcje związane ze środowiskiem wykonania. Plik nagłówkowy `omp.h` deklaruje dwa typy, funkcje użyteczne dla kontrolowania środowiska wykonywania równoległego, oraz funkcje blokujące, służące synchronizacji dostępu do danych.

Oto niektóre funkcje związane ze środowiskiem wykonywania równoległego:

— `omp_set_num_threads` — ustawia liczbę wątków używanych podczas wykonywania obszaru równoległego; działanie jest zależne od tego, czy umożliwione jest dynamiczne przydzielanie wątków — jeśli nie, to ustawiona wartość oznacza liczbę wątków tworzoną przy wejściu w każdy obszar równoległy (także zagnieżdżony); w przeciwnym wypadku wartość oznacza maksymalną liczbę wątków która może zostać użyta;

— `omp_get_num_threads` — funkcja zwraca aktualną liczbę wątków w grupie, wykonujących obszar równoległy, z którego wywołano funkcję;

— `omp_get_max_threads` — zwraca maksymalną wartość, jaka może zostać zwrócona po wywołaniu funkcji `omp_get_num_threads`.

Listing 8.2. Drugi przykład użycia pragm OpenMP

```
int WyznaczMPQ(int n, int *m, int *p, int *q) {
//Funkcja na podstawie zmiennej n
//wyznacza m, p, q,
//przy czym pod i suf to pewne dane funkcje
#pragma omp parallel sections shared (n)
  {
#pragma omp section
    *m = pod((n-1)/2.0);
#pragma omp section
    *p = pod((n+1)/2.0);
#pragma omp section
    *q = suf((n+1)/2.0);
  } //end pragma omp
  return 0;
```

```
} //end WyznaczMPQ
```

Funkcja `WyznaczMPQ` oblicza wartości trzech zmiennych: `m`, `p`, `q`. Za pomocą dyrektyw `sections` i `section` programista rozdziela pracę między trzy wątki. Jeden z wątków oblicza wartość zmiennej `m`, drugi — zmiennej `p`, trzeci — zmiennej `q`. Obliczenia te będą wykonywane równolegle, przez co czas obliczeń ulegnie znacznemu skróceniu[1].

Fragmenty programu zawarte w różnych sekcjach muszą być od siebie niezależne. Obliczenia wykonywane przez jeden z wątków nie mogą jednocześnie odwoływać się do obliczeń wykonywanych przez wątek drugi. W przypadku, gdy mamy do dyspozycji tylko jeden wątek obliczenia zawarte po dyrektywie `sections` zostaną przeprowadzone sekwencyjnie.

8.3. Model programowania z pamięcią rozproszoną

Do programowania komputerów z pamięcią rozproszoną wykorzystywany jest model procesorów komunikujących się poprzez wymianę komunikatów. Komputery wchodzące w skład sieci porozumiewają się ze sobą przez przesyłanie komunikatów.

Własności architektury wielu komputerów połączonych w sieć, które muszą być uwzględnione przy programowaniu w tym modelu to:
— każdy z procesorów ma dostęp tylko do swojej lokalnej pamięci;
— procesory komunikują się ze sobą poprzez system wejścia/wyjścia, a dalej poprzez sieć połączeń.

Każdy komputer wchodzący w skład sieci komputerów nazywany jest węzłem.

Podstawowe funkcjonalności wymagane w modelu z pamięcią rozproszoną to:
— utworzenie procesu na innym węźle;
— przesyłanie i odbiór komunikatów pomiędzy procesorami;
— narzędzia administracji węzłami.

Podstawowe operacje realizujące komunikację to — w zapisie umownym — **send**(*komunikat*) i **receive**(*komunikat*). Istnieje znaczne rozróżnienie typów komunikacji ze względu na *synchronizację*. Każda z instrukcji może wstrzymywać dalsze wykonywanie procesu, w którym została wywołana. Określa się ją mianem *synchronizacji blokującej*, w przeciwnym wypadku jest ona *synchronizacją nieblokującą* (*komunikacja!asynchroniczna*). W komunikacji blokującej **send**() wstrzymuje proces nadawczy do czasu wykonania **receive**() w procesie odbiorczym. Następuje wówczas przekazanie ko-

[1] Właściwie byłoby tak, gdyby były to dłuższe obliczenia — przy takich krótkich zrównoleglenie może być nieopłacalne.

munikatu, po czym oba procesy są kontynuowane — ten typ komunikacji nosi nazwę *komunikacji synchronicznej*.

Standardem w programowaniu równoległym korzystającym z rozproszonego systemu pamięci jest MPI. Zaletami MPI są:
— prędkość,
— skalowalność,
— przenośność — zaprojektowany dla właściwie każdej architektury opartej na pamięci rozproszonej.

8.3.1. MPI — specyfikacja interfejsu przesyłania wiadomości

Jednym z najważniejszych zagadnień w przetwarzaniu i programowaniu równoległym (w szczególności w programowaniu rozproszonym) jest komunikacja między procesami. Do przesyłania danych między procesorami wykorzystywany jest standard MPI (ang. *Message Passing Interface*) [38].

Nazwy wszystkich funkcji i stałych w MPI rozpoczynają się przedrostkiem MPI [47, 48]. Pierwszą procedurą, którą należy wywołać w programie równoległym jest procedura inicjująca, która ma postać [30]:

```
int MPI_Init(int *argc, char **argv)
```

Procedura ta akceptuje parametry podane do programu głównego, wywoływana jest przed jakąkolwiek inną procedurą i dokładnie raz na początku programu.

Procedurą kończącą program MPI jest procedura:

```
int MPI_Finalize(void)
```

Procedura ta czyści wszystkie struktury danych oraz kończy całą komunikacje w programie.

Prawie każda funkcja MPI zwraca liczbę całkowitą, która jest kodem błędu. Jeśli funkcja wykryje błąd to zwraca liczbę przy pomocy, której możemy zidentyfikować jego powód. Większość implementacji MPI przerywa wykonanie programu gdy taki błąd powstanie.

Ważnym pojęciem w MPI jest *komunikator*. `MPI\_COMM\_WORLD` jest stałą MPI oznaczającą komunikator, w którym znajdują się wszystkie procesy wywołujące funkcję `MPI\_Init`. W każdym komunikatorze procesy są numerowane od 0 i każdy proces ma unikalny numer. W programie rozproszonym może być kilka komunikatorów. Jeden proces może wchodzić w skład kilku komunikatorów i może mieć w każdym z nich inny numer.

Kolejne funkcje odpowiadają za odczytanie liczby procesorów w grupie oraz otrzymanie numer procesu w obrębie komunikatora:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Argument `comm` jest komunikatorem. Wskaźnikami do zmiennych przechowujących liczbę procesów oraz numer procesu są `size` oraz `rank`.

W MPI komunikacja [8, 38] jest bardzo rozbudowana. Dzielimy ją na komunikację: punkt–punkt i komunikację grupową. W komunikacji punkt–punkt rozróżniamy wysyłanie i odbieranie blokujące i wysyłanie i odbieranie nieblokujące. W komunikacji grupowej nie rozróżniamy oddzielnie wysyłania i odbierania.

8.3.1.1. Komunikacja punkt–punkt

Funkcja do wysyłania wiadomości w MPI wygląda następująco:

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)
```

Tabela 8.1. Predefiniowane typy MPI a typy języka C

MPI	C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	—
MPI_PACKED	—

Zawartość wiadomości jest przechowywana w bloku pamięci o adresie `buf`, inaczej można powiedzieć, że jest to adres danych wysyłanych. Parametr `count` mówi ile danych z `buf` typu `datatype` ma zostać wysłanych. Ponadto pozwalają one systemowi określić ilość miejsca potrzebną na wiadomość. `MPI_Datatype` jest to typ danych, które będą wysłane. Odpowiadają one w większości typom języka C, co ukazuje Tabela 8.1. Numer procesu w komunikatorze `comm`, do którego wysyłamy, określa `dest`. W celu odróżnienia różnych wiadomości, parametr `tag` pozwala na nadanie wiadomości

unikalnej etykiety, gdyż może się zdarzyć sytuacja, że proces otrzyma wiele wiadomości od jednego procesu w nieustalonej kolejności.

`MPI_Send` jest funkcją blokującą — oznacza to, że będzie blokować program dopóki wiadomości i etykieta nie zostanie bezpiecznie przechowana tak, by wysyłający proces mógł ją znowu modyfikować. W tym celu może być bezpośrednio skopiowana do bufora wysyłającego bądź do tymczasowego bufora systemowego. Blokujące wysyłanie może zostać ukończony od razu, gdy wiadomość zostanie zbuforowana — nawet, gdy nie została wykonana operacja odbioru. To powoduje, że może być ona kosztowna, gdyż wymaga kopiowania z pamięci do pamięci oraz alokacji pamięci na zbuforowanie. MPI oferuje kilku trybów komunikacji blokującej.

Standardowy — zakończenie wysyłania następuje w momencie, gdy bufor od nowa zacznie wysyłać kolejne porcje informacji. Nie wynika z tego, że wiadomość doszła, może ona zostać w sieci. Zakończenie wysyłania oznacza, że bufor może zostać ponownie użyty. Procedura realizująca standardowe blokujące wysyłanie wiadomości to `MPI_Send()`.

Synchroniczny — proces wysyłający musi wiedzieć, że wiadomość jest pobrana przez odbierający proces. Oba procesy muszą używać komunikacji synchronicznej. Proces odbierający wysyła do procesu wysyłającego informację potwierdzenia odbioru. Kończy się tylko wtedy, gdy zakończyło się odbieranie wiadomości. Procedura realizująca synchroniczne blokujące wysyłanie informacji to: `MPI_Ssend()` posiadająca parametry dokładnie takie same jaki w przypadku opisanej procedury `MPI_Send()`. Zaletą tej procedury jest fakt, że jest bezpieczniejsza, ale też wolniejsza od pozostałych procedur wysyłających.

Buforowany — gwarantuje natychmiastowe zakończenie procesu wysyłania, ponieważ kopiuje wiadomość do bufora systemowego. Procedura realizująca buforowane blokujące wysyłanie informacji to: `MPI_Bsend()` posiadająca parametry dokładnie takie same jaki w przypadku opisanej procedury `MPI_Send()`. Zaletą tej procedury jest wyskakiwanie błędów jeśli sieć jest zapchana, a wadą, że musi być wyraźnie zarezerwowany bufor przez programistę.

Gotowy — wysłanie kończy się natychmiastowo i proces wysyłający ma nadzieję, że proces odbierający był gotowy do odebrania wiadomości, jeśli nie – dane są gubione. Procedura realizująca gotowe blokujące wysyłanie wiadomości to `MPI_Rsend()`.

Dla polepszenia wydajność można oddzielić komunikację od obliczeń i uczynić by się wzajemnie przeplatały. W tym celu można zastosować wątki bądź inny rodzaj komunikacji – komunikację nieblokującą. Wywołanie takiej komunikacji po prostu ją rozpoczyna, ale jej nie kończy, nie blokuje programu. To program kończy komunikację na późniejszym etapie wywołując odpowiednią funkcję.

Funkcja, która pozwala odebrać wiadomość to:

```
int MPI_Recv(void* buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Status *status)
```

Funkcja ta odbiera dane w ilości `count` do bufora `buf`. Dodatkowo `source` jest numerem procesu wysyłającego dane. Można także nie specyfikować numeru procesu wysyłającego i etykiety. Do tego celu służą stałe MPI: `MPI_ANY_SOURCE` i `MPI_ANY_TAG`. Ostatni parametr status zwraca informacje o danych, które zostały odebrane. Jest to struktura zawierająca: źródło, etykietę oraz kod błędu. Przedstawia się następująco

```
status -> MPI_SOURCE
status -> MPI_TAG
status -> MPI_ERROR
```

W celu zignorowania tego argumentu można posłużyć się predefiniowanymi stałymi: `MPI_STATUS_IGNORE` i `MPI_STATUSES_IGNORE`.

Funkcja `MPI_Recv` jest funkcją blokującą pozwalającą na odebranie wiadomości. Zostaje ona wykonana tylko po tym jak odbierający bufor ma nową wiadomość. Wstrzymuje ona program (blokuje go) do czasu otrzymania nowej wiadomości.

Procedury do komunikacji nieblokującej są takie same jak procedury do komunikacji blokującej, poza tym, że posiadają dodatkowy – ostatni – parametr nazywany `request` pozwalający na testowanie zakończenia komunikacji. Podstawowymi funkcjami w MPI dla nieblokującej komunikacji są `MPI_Isend` oraz `MPI_Irecv` mające poniższą postać.

```
int MPI_Isend(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void* buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Request *request)
```

Parametry wspólne z `MPI_Send` i `MPI_Recv` mają takie samo znaczenie. Rozpoczęcia `MPI_Isend` jest informacją dla systemu, że może zacząć kopiować do bufora wysyłającego (bufora systemowego lub do odbiorcy). Funkcja `MPI_Irecv` znaczy, że system może zacząć kopiować dane do bufora. Te bufory nie powinny być modyfikowane dopóty operacje nie zostaną ukończone. Komunikacja nieblokująca używa obiektów `request`, które pozwalają zidentyfikować operacje komunikacyjne oraz w połączeniu z odpowiednimi operacjami – przerwać je. Są to systemowe obiekty do których mamy dostęp poprzez wskaźnik, aczkolwiek nie jest to dostęp bezpośredni. Taki obiekt

identyfikuje różne własności takiej operacji jak na przykład tryb, bufor, etykietę. Po ukończeniu operacji pozwala na zidentyfikowanie zakończonej operacji.

Do sprawdzenia czy komunikacja już się zakończyła używamy procedur testujących: `MPI_Wait` oraz `MPI_Test`.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Test(MPI_Request *request, int *flag,
  MPI_Status *status)
```

Ukończenie operacji wysyłającej oznacza, że wysyłający proces może modyfikować bufor, nie musi jednak oznaczać, że wiadomość została odebrana. W odniesieniu do odebrania wiadomości ukończenie operacji oznacza, że bufor zawiera nową wiadomość. `MPI_Wait` czeka aż operacja identyfikowana przez request się zakończy; w przypadku gdy była ona nieblokująca wskaźnik request jest ustawiony na `MPI_REQUEST_NULL`. Natomiast `MPI_TEST` sprawdza czy operacja określona przez `request` została zakończona i zwraca kod powrotu.

Funkcje, które mogą być przydatne do kończenia komunikacji:

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
  int *flag, MPI_Status *status)
int MPI_Probe(int source, int tag, MPI_Comm comm,
  MPI_Status *status)
int MPI_Cancel(MPI_Request *request)
```

`MPI_Iprobe` zwraca `flag==true` jeżeli jest wiadomość, która może być odebrana i pasuje do wzorca wyspecyfikowanego przez `source`, `tag` oraz `comm`. Nie ma konieczności odebrania wiadomości po jej próbkowaniu i ta sama wiadomość może być próbkowana wiele razy. Funkcja `MPI_Probe` jest podobna do poprzedniej. Jednakże blokuje ona program i zwraca sterowanie dopiero wtedy gdy wiadomość została znaleziona. Ostatnia funkcja pozwala na anulowanie trwającej nieblokującej komunikacji. Zwraca sterowanie od razu, prawdopodobnie przed anulowaniem komunikacji. Stąd istnieje potrzeba ukończenia komunikacji funkcjami takimi jak `MPI_Wait` czy `MPI_Test`.

8.3.1.2. Komunikacja grupowa

Komunikacja grupowa obejmuje grupę lub grupy procesów. Często w komunikacji grupowej wyróżniony jest jeden proces zwany *rootem* i to on rozsyła lub zbiera wszystkie wiadomości od innych procesów. Komunikacja grupowa odbywa się w obrębie zdefiniowanego komunikatora. W komunikacji grupowej nie ma podziału na procedury wysyłające i odbierające, one jednocześnie wysyłają do kogo trzeba i odbierają. MPI dostarcza także operacji ustawienia bariery, która synchronizuje wszystkie procesy w komunikatorze.

Wywołanie operacji `MPI_Barrier()` powoduje wstrzymanie pracy procesów, aż do momentu, kiedy wszystkie procesy związane z komunikatorem *comm* dojdą do miejsca wywołania tej procedury. Następnie wszystkie razem mogą rozpocząć dalsze wykonywanie programu.

Najważniejszymi operacjami do komunikacji grupowej są:

— `MPI_Barrier` — tworzy barierę synchronizującą procesy w komunikatorze;
— `MPI_Bcast` — pojedynczy proces wysyła te same dane do każdego procesu w komunikatorze;
— `MPI_Gather` — gromadzi dane od wszystkich procesów do jednego procesu;
— `MPI_Scatter` — rozprasza dane od jednego procesu do wszystkich procesów; wysyłana informacja może być różna dla każdego procesu, porcje wysyłanych informacji są takiej samej wielkości dla każdego procesu;
— `MPI_Allgather` — procedura ta wysyła informację od każdego procesu do każdego; wielkość wysyłanej informacji jest taka sama dla wszystkich procesów, wszystkie procesy w komunikatorze `comm` są jednocześnie wysyłające i odbierające, po wykonaniu tej procedury każdy proces posiada taką samą informację; w tej procedurze nie ma wyróżnionego procesu root;
— `MPI_Alltoall` — procedura ta wysyła informację od każdego procesu do każdego; wiadomość wysyłana do każdego procesu ma taką samą wielkość, ale inną zawartość, wszystkie procesy w komunikatorze `comm` są jednocześnie wysyłające i odbierające, po wykonaniu tej procedury procesy posiadają różne informacje; w tym typie komunikacji nie ma wyróżnionego procesu root;
— `MPI_Allreduce` oraz `MPI_Reduce` — wykonują operacje redukcyjne i wysyłają wynik do wszystkich procesów.

Przedstawimy tu i omówimy operacje redukcyjne.

```
int MPI_Reduce(void* sendbuf, void* recvbuf,
  int count,MPI_Datatype datatype, MPI_Op op,
  int root, MPI_Comm comm)
```

Wykonuje ona operację redukcyjną zdefiniowaną w zmiennej `op` (Tabela 8.2) na danych wejściowych umieszczonych w `sendbuf` zapisując wynik w procesie `root` posługując się `recvbuf`. Gdy procesy wyliczają jakąś wartość to można je wysłać i od razu zredukować w procesie `root`. Zmodyfikowany wariant tej procedury to:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype, MPI_Op op,
  MPI_Comm comm)
```

która zredukowany wynik umieszcza we wszystkich procesach w obrębie komunikatora `comm` w buforze `recvbuf`.

Tabela 8.2. Znaczenie operatorów MPI

stała	znaczenie
MPI_MAX	maksimum
MPI_MIN	minimum
MPI_SUM	suma
MPI_PROD	iloczyn
MPI_LAND	logiczne i
MPI_BAND	bitowe i
MPI_LOR	logiczne lub
MPI_BOR	bitowe lub
MPI_LXOR	logiczne albo
MPI_BXOR	bitowe albo
MPI_MAXLOC	maksymalna wartość i położenie
MPI_MINLOC	minimalna wartość i położenie

8.3.1.3. Kompilacja i uruchamianie

Kompilacja oraz uruchomienie programu napisanego z wykorzystaniem standardu MPI są specyficzne. Standardowo program MPI dla języka C/C++ kompiluje się poleceniem:

```
mpicc -o program plik_zrodlowy.c
```

Program jest uruchamiany poleceniem `mpirun` z parametrem określającym liczbę procesorów `-np` (ang. *number of processors*)

```
mpirun -np 4 ./program argumenty
```

(aplikacja `program` będzie uruchomiona na 4 procesorach).

Listing 8.3. Pierwszy przykład użycia MPI

```
1 #include <stdio.h>
  #include "mpi.h"
3
  main(int argc, char** argv) {
5   int mojnumer;
    int liczproc;
7   char wiadomosc[100];
    char info[100];
9   int dest;
    int tag = 50;
11  int i;
    MPI_Status status;
13
```

```
    MPI_Init(&argc, &argv);
15    //Procedura zwraca numer procesu wywolujacego
      //procedure
17  MPI_Comm_rank(MPI_COMM_WORLD, &mojnumer);
      //Procedura zwraca ilosc procesow wchodzacych
19    //w sklad komunikatora
    MPI_Comm_size(MPI_COMM_WORLD, &liczproc);
21
    if (mojnumer!=0) {
23    sprintf(wiadomosc,"Powitanie od procesu %d.",
          mojnumer);
25      //dest - numer procesu odbierajacego
      dest = 0;
27    MPI_Send(wiadomosc,100, MPI_CHAR, dest, tag,
          MPI_COMM_WORLD);
29  } else {
      for(i = 1;i < liczproc; i++) {
31      //pobranie wiadmosci od procesu o numerze
        //i zwrocenie parametru status
33      MPI_Recv(info,100, MPI_CHAR,i, tag,
            MPI_COMM_WORLD, &status);
35      printf("Od %d, tresc wiadomosci: %s\n", i, info);
      } //for
37  } //else
    MPI_Finalize();
39 } //main
```

Program rozpoczyna się deklaracją używanych zmiennych i dalej następuje wywołanie procedury inicjującej, następnie uzyskujemy informacje na temat procesów wchodzących w skład komunikatora `MPI_COMM_WORLD`. Dalej, przy pomocy selekcji badamy czy numer procesu ma wartość 0 — jeśli tak, odbiera on wiadomości od pozostałych procesów w komunikatorze i wyświetla tę informację na ekranie. Jeśli jest to proces o numerze różnym od zera, nadaje on wartość zmiennej wiadomosc i wysyła ją do procesu o numerze 0.

Po uruchomieniu programu równoległego poleceniem:
`mpirun -np 5 ./program`
na konsoli pojawi się tekst:

```
Od 1, tresc wiadomosci: Powitanie od procesu 1.
Od 2, tresc wiadomosci: Powitanie od procesu 2.
Od 3, tresc wiadomosci: Powitanie od procesu 3.
Od 4, tresc wiadomosci: Powitanie od procesu 4.
```

Listing 8.4. Drugi przykład użycia MPI

```
1 #include <stdio.h>
```

```
#include "mpi.h"

main(int argc, char** argv) {
  int mojnumer;
  int liczba=15;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mojnumer);
  liczba+=mojnumer;
  MPI_Bcast(&liczba,1, MPI_INT, 4, MPI_COMM_WORLD);
  printf("Moj numer %d, otrzymana liczba to: %d\n",
      mojnumer, liczba);
  MPI_Finalize();
} //main
```

Program rozpoczyna się od deklaracji zmiennych. Zmienna mojnumer określa numer procesu, zmienna liczba jest typu całkowitego. W treści podprogramu zmieniamy wartość zmiennej liczba, powiększając jej wartość o wartość zmiennej mojnumer. Następnie proces o numerze 4 rozsyła wartość zmiennej liczba do wszystkich procesów w komunikatorze `MPI_COMM_WORLD`. Następnie każdy z procesorów wyświetla na ekranie informację o swoim numerze i wartości otrzymanej liczby. Po skompilowaniu i uruchomieniu na 5 procesorach na konsoli wyglądać może to w następujący sposób:

```
Moj numer 4, otrzymana liczba to: 19
Moj numer 3, otrzymana liczba to: 19
Moj numer 0, otrzymana liczba to: 19
Moj numer 2, otrzymana liczba to: 19
Moj numer 1, otrzymana liczba to: 19
```

8.4. Pytania i zadania

1. Wyjaśnij pojęcia: węzeł, komunikacja synchroniczna, komunikacja asynchroniczna, komunikator, pragma, bariera.
2. Jakim poleceniem uruchomić program równoległy z Listingu 8.1 na 5, 10 i 15 procesorach?
3. Określ zakres zmiennych `mojnumer` i `liczba` z Listingu 8.2.
4. Napisz co się stanie jeśli program z Listingu 8.4 zostanie uruchomiony poleceniem:
   `mpirun -np 3 ./program`
5. Program z przykładu II został zmodyfikowany, jak to pokazano na Listingu 8.5.

Listing 8.5. Zmodyfikowany przykład MPI

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv) {
  int mojnumer;
  int liczba = 15;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mojnumer);
  MPI_Bcast(&liczba, 1, MPI_INT, 3, MPI_COMM_WORLD);
  liczba += mojnumer;
  printf("Moj numer: %d, otrzymana liczba to: %d.\n",
      mojnumer, liczba);
  MPI_Finalize();
}
```

Po skompilowaniu i uruchomieniu na 5 procesorach wpisz w miejsce kropek brakujące liczby, które pojawią się podczas uruchomienia tego programu na konsoli.

```
Moj numer: ..., otrzymana liczba to: ...
Moj numer: ..., otrzymana liczba to: ...
Moj numer: ..., otrzymana liczba to: ...
Moj numer: ..., otrzymana liczba to: ...
Moj numer: ..., otrzymana liczba to: ...
```

6. Porównaj modele programowania równoległego z pamięcią wspólną i z pamięcią rozproszoną.
7. Czym różnią się paradygmaty programowania: sekwencyjny, współbieżny, równoległy, rozproszony?
8. Scharakteryzuj elementy pożądane w języku przystosowanym do programowania maszyn wieloprocesorowych.
9. Scharakteryzuj elementy pożądane w języku przystosowanym do programowania rozproszonego.
10. Wyjaśnij różnice między komunikacją blokującą a nieblokującą.
11. Wyjaśnij różnice między komunikacją punkt–punkt, a komunikacją grupową.
12. Co to jest operacja redukcji?

Bibliografia

[1] D. Adams: *Autostopem przez Galaktykę*, Zysk i S-ka, Poznań 1996.

[2] A. V. Aho, R. Sethi, J. D. Ullman: *Kompilatory. Regyły, metody i narzędzia*, WNT, Warszawa 2002.

[3] H. Aït-Kaci: *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, MA 1991.
`http://www.cvc.uab.es/shared/teach/a25002/wambook.pdf`[2]

[4] J. Armstrong: w: *Coders at Work: Reflections on the Craft of Programming*, P. Seibel, Apress 2009.

[5] J. W. Backus: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, w: *Proceedings of the International Conference on Information Processing. UNESCO*, 1959, strony 125–132,
`http://www.softwarepreservation.org/projects/ALGOL/paper/▷`
`▷Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf`

[6] H. Barendregt: *Lambda Calculi with Types*.
`ftp://ftp.cs.ru.nl/pub/CompMath.Found/HBK.ps`

[7] H. Barendregt: The Type Free Lambda Calculus, w: *Handbook of Mathematical Logic*, North-Holland 1977, strony 1091–1132.

[8] B. Bylina: Komunikacja w MPI, w: *Informatyka Stosowana S2/2001, V Lubelskie Akademickie Forum Informatyczne, Kazimierz Dolny* 2001, strony 31–40.

[9] J. Cain: *Programming paradigms*.
`http://see.stanford.edu/see/courseinfo.aspx?▷`
`▷coll=2d712634-2bf1-4b55-9a3a-ca9d470755ee`

[10] A. Denisjuk: *Sztuczna inteligencja. Programowanie w Prologu*.
`http://free.of.pl/p/prolog/strona2/data/zagadki.pdf`

[11] E. W. Dijkstra: Go To Statement Considered Harmful, w: *Communications of the ACM*, Vol. 11, No. 3, March 1968, strony 147–148.
`http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF`

[12] A. B. Downey: *Python for Software Design: How to Think Like a Computer Scientist*, Cambridge University Press 2009.
`http://greenteapress.com/thinkpython/thinkpython.html`

[13] S. Ferg: *Python & Java: A Side-by-Side Comparison*.
`http://pythonconquerstheuniverse.wordpress.com/category/▷`
`▷java-and-python/`

[2] Wszystkie zamieszczone w bibliografii adresy internetowe były poprawne i aktywne w listopadzie 2010.

[14] R. W. Floyd: Assigning meanings to programs, w: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, Vol. 19, 1967, strony 19–31.
`http://www.cs.virginia.edu/~weimer/2007-615/reading/▷`
`▷FloydMeaning.pdf`

[15] I. Foster: *Designing and Building Parallel Programs*, Addison Wesley 1996.
`http://www.mcs.anl.gov/dbpp`

[16] P. Graham: *Why ARC isn't especially Object–Oriented*.
`http://www.paulgraham.com/noop.html`

[17] Z. Grodzki, J. Kowalska: *Zbiór zadań z lingwistyki matematycznej*, Wydawnictwo UMCS, Lublin 1993.

[18] G. T. Heineman, W. T. Councill: *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley Professional, Reading 2001.

[19] R. Hettinger: *Descriptor HowTo Guide*.
`http://docs.python.org/dev/howto/descriptor.html`

[20] C. A. R. Hoare: An axiomatic basis for computer programming, w: *Communications of the ACM*, 12 (10) Oct. 1969, strony 576–583.
`http://sunnyday.mit.edu/16.355/Hoare-CACM-69.pdf`

[21] C. A. R. Hoare, N. Wirth: An axiomatic definition of the programming language Pascal, w: *Acta Informatice* 2, 1973, strony 335–355.

[22] J. H. Holland, J. H. Miller: Artificial Adaptive Agents in Economic Theory, w: *American Economic Review* 81 (2), 1991, strony 365–371.
`http://zia.hss.cmu.edu/miller/papers/aaa.pdf`

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming, w: *Proceedings of the European Conference on Object-Oriented Programming*, vol. 1241, 1997, strony 220–242.

[24] R. Kowalski: Predicate Logic as a Programming Language, w: *Proceedings IFIP Congress*, Stockholm, North Holland Publishing Co. 1974, strony 569–574.
`http://www.doc.ic.ac.uk/~rak/papers/IFIP 74.pdf`

[25] R. Kowalski: The Early Years of Logic Programming, w: *Comm. of ACM*, Vol. 31, n. 1, Jan. 1988.
`http://www.doc.ic.ac.uk/~rak/papers/the early years.pdf`

[26] R. Kowalski, D. Kuehner: Linear Resolution with Selection Function, w: *Artificial Intelligence*, Vol. 2, 1971, strony 227–260.
`http://www.doc.ic.ac.uk/~rak/papers/sl.pdf`

[27] A. M. Kuchling: *Functional Programming HOWTO*.
`http://docs.python.org/dev/howto/functional.html`

[28] K. Liebherr: *Adaptive Object Oriented Programming: The Demeter Approach with Propagation Patterns*, 1996.

[29] M. Lutz, D. Ascher: *Python. Wprowadzenie*, Helion, Gliwice 2002.

[30] Ł. Łapiński: *Porównanie efektywności wybranych algorytmów rozwiązywania układów równań w wersji synchronicznej i asynchronicznej przy użyciu MPI*, praca magisterska, UMCS, Lublin 2010.

[31] R. Mansfield: *Has OOP Failed?*.
`http://www.scribd.com/doc/35164145/Has-OOP-FailedY`

[32] R. Mansfield: *OOP Is Much Better in Theory Than in Practice*.
`http://www.devx.com/DevX/Article/26776`
[33] A. Martelli, A. Martelli Ravenscroft, D. Ascher: *Python. Receptury*, Helion, Gliwice 2006.
[34] J. Martinek: *Lisp. Opis, realizacja i zastosowanie*, WNT, Warszawa 1980.
[35] J. McCarthy: Programs with common sense, w: *Symposium on Mechanization of Thought Processes*, National Physical Laboratory, Teddington, England 1958.
`http://www-formal.stanford.edu/jmc/mcc59.pdf`
[36] B. Meyer: *Design by Contract*, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc. 1986.
[37] M. Moczurad, W. Moczurad: *Paradygmaty programowania*.
`http://wazniak.mimuw.edu.pl/index.php?▷`
`▷title=Paradygmaty_programowania`
[38] *MPI: A Message-Passing Interface Standard*, wersja 2.2, 2009.
`http://www.lrz.de/services/software/parallel/mpi/▷`
`▷mpi22-report.pdf`
[39] J. Mycka: *Semantyka języków programowania*.
`http://hektor.umcs.lublin.pl/~jerzm/strony/semantyka/semantyka.html`
[40] J. Mycka: *Teoria funkcji rekurencyjnych*. Lublin 2000.
`http://hektor.umcs.lublin.pl/~jerzm/strony/frekurencyjne/▷`
`▷skrypt.ps`
[41] T. Niemann: *A Compact Guide to Lex & Yacc*, epaperpress.com.
`http://epaperpress.com/lexandyacc/download/lexyacc.pdf`
[42] U. Nilsson, J. Maluszynski: *Logic, Programming and Prolog (2ed)*, 2000.
`http://www.ida.liu.se/~ulfni/lpp/`
[43] P. Norton i inni: *Python. Od podstaw*, Helion, Gliwice 2006.
[44] H. Oktaba, W. Ratajczak: *Simula 67*, WNT, Warszawa 1980.
[45] *OpenMP C/C++ API*, wersja 3.0, 2008.
`http://www.openmp.org/`
[46] B. O'Sullivan, J. Goerzen, D. Stewart: *Real World Haskell*, O'Reilly 2008.
[47] P. S. Pacheco: *A user's guide to MPI*, 1998.
`ftp://math.usfca.edu/pub/MPI/mpi.guide.ps`
[48] P. S. Pacheco:*Parallel programming with MPI*, Morgan Kaufmann 1997.
[49] A. Paluszek: *Paradygmaty programowania obiektowego i aspektowego — studium porównawcze*, praca magisterska, UMCS, Lublin 2010.
[50] M. Pilgrim: *Zanurkuj w Pythonie*.
`http://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie`
[51] P. van Roy: Programming Paradigms for Dummies: What Every Programmer Should Know, w: *New Computational Paradigms for Computer Music*, redakcja: G. Assayag, A. Gerzso, IRCAM/Delatour, France 2009.
`http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf`
[52] P. van Roy, S. Haridi: *Concepts, Techniques, and Models of Computer Programming*, MIT Press 2004.
[53] A. Salomaa: *Formal Languages*, Academic Press, New York 1973.
[54] M. L. Scott: *Programming Laguage Pragmatics. Second Edition*, Elsevier/Morgan Kaufmann 2006.

[55] R. Sebesta: *Concepts of Programming Languages*, Addison Wesley 2005.
[56] P. Stpiczyński, M. Paprzycki: A brief introduction to parallel computing, w: *Handbook of parallel computing and statistics*, redakcja: E. J. Kontoghiorghes, Chapman & Hall/CRC 2006.
[57] M. Summerfield: *Moving from Python 2 to Python 3*.
`http://ptgmedia.pearsoncmg.com/imprint_downloads/informit/▷`
`▷promotions/python/python2python3.pdf`
[58] M. M. Sysło: *Algorytmy*, WSiP, Warszawa 1997.
[59] S. Thompson: *Haskell. The Craft of Functional Programming*, Addison Wesley 1999.
[60] D. A. Turner: An Overview of Miranda, w: *Research Topics in Funtional Programming*, redakcja: D. A. Turner, Addison Wesley 1990.
[61] W. M. Turski: *Struktury danych*, WNT, Warszawa 1976.
[62] P. Wadler: Monads for functional programming, w: *Advanced Functional Programming*, LNCS 925, Springer Verlag 1995.
[63] W. M .Waite, G. Goos: *Konstrukcja kompilatorów*, WNT, Warszawa 1989.
[64] D. H. D. Warren: An abstract Prolog instruction set, w: *Technical Note 309*, SRI International, Menlo Park, CA, Oct. 1983.
`http://www.ai.sri.com/pubs/files/641.pdf`
[65] N. Wirth: *Algorytmy + struktury danych = programy*, WNT, Warszawa 2001.
[66] N. Wirth: Good Ideas, Through the Looking Glass, w: *IEEE Computer*, Jan. 2006, stony 56–68.
`http://www.inf.ethz.ch/personal/wirth/Articles/GoodIdeas.pdf`
[67] N. Wirth: What Can We Do about the Unnecessary Diversity of Notations for Syntax Definitions? w: *Communications of the ACM*, 20 (11) Nov. 1977, strony 822–823.
[68] Ś. Ząbek: *Strukturalne techniki programowania*, Wydawnictwo UMCS, Lublin 2004.
[69] `http://docs.python.org/tutorial/`
[70] `http://en.wikipedia.org/wiki/Ada_Lovelace`
[71] `http://en.wikipedia.org/wiki/Alonso_Church`
[72] `http://en.wikipedia.org/wiki/Barbara_Liskov`
[73] `http://en.wikipedia.org/wiki/Duck_typing`
[74] `http://en.wikipedia.org/wiki/Grace_Hopper`
[75] `http://en.wikipedia.org/wiki/Haskell_Curry`
[76] `http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)`
[77] `http://en.wikipedia.org/wiki/Parallel_computing`
[78] `http://en.wikipedia.org/wiki/Von_Neumann_architecture`
[79] `http://mail.python.org/pipermail/tutor/2003-October/025932.html`
[80] `http://oakwinter.com/code/functional/`
[81] `http://python.org/doc/`
[82] `http://www.dabeaz.com/ply/`
[83] `http://www.sjp.pl/co/paradygmat`
[84] `http://www.swi-prolog.org/pldoc/index.html`

Skorowidz