Wydanie II

Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman

KOMPILATORY

REGUŁY • METODY • NARZĘDZIA

PWN

KOMPILATORY

Wydanie II

Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman

KOMPILATORY

REGUŁY • METODY • NARZĘDZIA

PWN

Dane oryginału
Authorized translation from the English language edition, entitled: COMPILERS: PRINCIPLES, TECHNIQUES, AND TOOLS; Second Edition; ISBN 0321486811; by Alfred V. Aho; and by Monica S. Lam; and by Ravi Sethi; and by Jeffrey D. Ullman; published by Pearson Education, Inc, publishing as Addison Wesley.

Przekład **Marek Włodarz** na zlecenie **WITKOM Witold Sikorski**

Konsultant **Janusz Majewski**

Projekt okładki i stron tytułowych **Joanna Andryjowicz**

Wydawca **Edyta Kawala**

Redaktor prowadzący **Jolanta Kowalczuk**

Redaktorzy merytoryczni **Janusz Majewski** (rozdz. 1–8)
Grzegorz Herman (rozdz. 9–12, dod. A i B)

Korekta **Anna Marecka**

Redaktor techniczny **Maria Czekaj**

Koordynator produkcji **Anna Bączkowska**

Skład i łamanie **FixPoint**

ISBN 978-83-01-20381-8

Wydanie II (I w WN PWN)
Warszawa 2019

Wydawnictwo Naukowe PWN SA
02-460 Warszawa, ul. Gottlieba Daimlera 2
tel. 22 69 54 321, faks 22 69 54 288
infolinia 801 33 33 88
e-mail: pwn@pwn.com.pl, reklama@pwn.pl
www.pwn.pl

Druk i oprawa Paper&Tinta

Spis treści

Przedmowa

W czasie, który upłynął od roku 1986, roku pierwszego wydania tej książki, świat projektowania kompilatorów znacząco się zmienił. Ewolucja języków programowania stworzyła nowe problemy. Architektury komputerów oferują dziś bogactwo zasobów, które projektant kompilatora powinien, a w zasadzie musi wykorzystać. Być może najbardziej interesujące jest to, że szanowane techniki optymalizowania kodu znalazły zastosowania poza kompilatorami. Są dziś używane w narzędziach wyszukujących błędy, a co najważniejsze, luki zabezpieczeń w już istniejącym oprogramowaniu. Zarazem większość technologii „przodowych" – gramatyki, wyrażenia regularne, parsery i translatory sterowane składnią – nadal jest w szerokim użyciu.

Tym samym nasza filozofia prezentowana w poprzednich wersjach tej książki się nie zmieniła. Zdajemy sobie sprawę, że bardzo nieliczni spośród czytelników będą tworzyć, lub choćby utrzymywać, kompilatory dla któregoś z głównych języków programowania. Jednak modele, teoria i algorytmy powiązane z kompilatorami mogą być stosowane w szerokim zakresie problemów projektowania i rozwijania oprogramowania. Dlatego szczególnie wyróżniamy kwestie do rozstrzygnięcia, które są najczęściej spotykane przy projektowaniu procesorów języków, niezależnie od języka źródłowego czy maszyny docelowej.

Korzystanie z książki

Opanowanie całości czy choć większości materiału z tej książki wymaga co najmniej dwóch kwartałów, a może nawet dwóch semestrów. Typowe podejście polega na przedstawieniu pierwszej połowy w ramach wykładu podstawowego, drugą zaś połowę tematyki książki – optymalizowanie kodu – na poziomie dyplomowym lub pośrednim. Oto konspekt poszczególnych rozdziałów:

W **rozdziale 1** zawarto materiały motywujące, a ponadto przedstawiono w nim kilka podstawowych zagadnień architektury komputerów i zasady języków programowania.

W **rozdziale 2** pokazano projektowanie miniaturowego kompilatora i wprowadzono wiele ważnych koncepcji, które zostaną rozwinięte w kolejnych rozdziałach. Sam kompilator został w całości zamieszczony w dodatku na końcu książki.

W **rozdziale 3** zawarto omówienie analizy leksykalnej, wyrażeń regularnych, automatów skończonych i narzędzi generujących leksery. Materiał ten jest podstawą do przetwarzania tekstów dowolnego rodzaju.

W **rozdziale 4** zaprezentowano główne metody parsingu: zstępujące (metoda zejść rekurencyjnych LL) i wstępujące (LR i jej warianty).

W **rozdziale 5** wprowadzono podstawowe koncepcje definicji kierowanych składnią i translacji sterowanej składnią.

W **rozdziale 6** rozwinięto teorię z rozdziału 5 i pokazano, jak można ją wykorzystać do generowania kodu pośredniego dla typowego języka programowania.

W **rozdziale 7** skupiono się na środowiskach wykonawczych, ze szczególnym naciskiem na zarządzanie stosem w czasie wykonania i mechanizmy odśmiecania pamięci.

W **rozdziale 8** omówiono generowanie kodu wynikowego. Obejmuje ono konstruowanie bloków podstawowych, generowanie kodu dla wyrażeń i bloków podstawowych oraz techniki alokowania rejestrów.

W **rozdziale 9** wprowadzono technologie optymalizacji kodu, w tym grafy przepływu, problemy przepływu danych i iteracyjne algorytmy rozwiązywania tych problemów.

W **rozdziale 10** zaprezentowano optymalizacje na poziomie instrukcji. Główny nacisk został tu położony na możliwości wydobycia równoległości z małych sekwencji instrukcji i szeregowania ich na pojedynczych procesorach, które są w stanie wykonywać więcej niż jedną czynność naraz.

W **rozdziale 11** omówiono wykrywanie i wykorzystywanie równoległości w większej skali. W tym miejscu skupiono uwagę na programach numerycznych mogących zawierać wiele ciasnych pętli przebiegających wielowymiarowe tablice.

W **rozdziale 12** zajęto się analizami międzyproceduralnymi. Omówiono tu analizy wskaźników, aliasowania oraz przepływu danych, uwzględniające sekwencje wywołań procedur, które osiągają dany punkt w kodzie.

Wykłady oparte na materiale zawartym w tej książce były prowadzone na uniwersytetach Columbia, Harvard i Stanford. Na Uniwersytecie Columbia regularnie proponowany jest wykład dla pierwszego roku studiów wyższego poziomu na temat języków programowania i translatorów, wykorzystujący materiał z pierwszych ośmiu rozdziałów. Cechę wyróżniającą tego wykładu stanowi trwający semestr projekt, w którym studenci w małych zespołach pracują nad utworzeniem i implementacją prostego języka własnego projektu. Tworzone przez nich języki obejmują wielką rozmaitość zastosowań, w tym obliczenia kwantowe, syntezowanie muzyki, grafikę komputerową, gry, operacje na macierzach i wiele innych obszarów. Do zbudowania swoich własnych kompilatorów studenci wykorzystują generatory komponentów kompilatorów, takie jak ANTLR, Lex lub Yacc, oraz techniki translacji sterowanej składnią omówione w rozdziałach 2 i 5. Następujący później zaawansowany wykład skupia się na materiałach rozdziałów 9–12, z wyróżnieniem generowania i optymalizacji kodu dla nowoczesnych maszyn, w tym procesorów sieciowych i architektur wieloprocesorowych.

Na Uniwersytecie Stanforda kwartalny wykład wprowadzający obejmuje w przybliżeniu materiał z rozdziałów 1–8, choć znajduje się w nim wprowadzenie do globalnych technik optymalizacyjnych z rozdziału 9. Drugi wykład obejmuje treść rozdziałów 9–12 oraz bardziej zaawansowany materiał dotyczący odśmie-

cania pamięci z rozdziału 7. Studenci wykorzystują opracowany na miejscu, oparty na Javie, system o nazwie Joeq do implementowania algorytmów analizy przepływu danych.

Wymagania wstępne

Czytelnik powinien dysponować pewnym „wyrafinowaniem informatycznym", obejmującym co najmniej zaawansowany wykład na temat programowania oraz wykłady ze struktur danych i matematyki dyskretnej. Przydatna będzie także znajomość kilku różnych języków programowania.

Ćwiczenia

W książce zawarto rozbudowane ćwiczenia, po kilka dla niemal każdego podrozdziału. Trudniejsze ćwiczenia lub ich części zostały wyróżnione wykrzyknikiem. Najtrudniejsze ćwiczenia oznaczono podwójnym wykrzyknikiem.

Podziękowania

Jon Bentley dostarczył wyczerpujących komentarzy o wielu rozdziałach wcześniejszego szkicu tej książki. Pomocne komentarze i poprawki dostarczyli również (w kolejności alfabetycznej): Domenico Bianculli, Peter Bosch, Marcio Buss, Marc Eaddy, Stephen Edwards, Vibhav Garg, Kim Hazelwood, Gaurav Kc, Wei Li, Mike Smith, Art Stamness, Krysta Svore, Olivier Tardieu oraz Jia Zeng. Jesteśmy bardzo wdzięczni za pomoc tych wszystkich osób. Wina za wszelkie pozostawione błędy oczywiście spoczywa na nas.

Dodatkowo Monica chciałaby podziękować swoim koleżankom i kolegom z zespołu kompilatora SUIF za 18-letnią naukę kompilowania, są to: Gerald Aigner, Dzintars Avots, Saman Amarasinghe, Jennifer Anderson, Michael Carbin, Gerald Cheong, Amer Diwan, Robert French, Anwar Ghuloum, Mary Hall, John Hennessy, David Heine, Shih-Wei Liao, Amy Lim, Benjamin Livshits, Michael Martin, Dror Maydan, Todd Mowry, Brian Murphy, Jerey Oplinger, Karen Pieper, Martin Rinard, Olatunji Ruwase, Constantine Sapuntzakis, Patrick Sathyanathan, Michael Smith, Steven Tjiang, Chau-Wen Tseng, Christopher Unkel, John Whaley, Robert Wilson, Christopher Wilson oraz Michael Wolf.

A.V.A., Chatham NJ
M.S.L., Menlo Park CA
R.S., Far Hills NJ
J.D.U., Stanford CA

Czerwiec 2006 r.

Rozdział 1

Wprowadzenie

Języki programowania są sposobami zapisu przestawiającymi obliczenia w sposób zrozumiały dla ludzi i dla maszyn. Świat, jaki dziś znamy, uzależniony jest od języków programowania, gdyż całe oprogramowanie działające na wszystkich komputerach zostało napisane w jakimś języku programowania. Jednak zanim możliwe będzie uruchomienie programu, musi on najpierw zostać przetłumaczony do postaci, w której komputer będzie mógł go wykonać.

Systemy programowe, które wykonują to tłumaczenie, nazywamy *kompilatorami*.

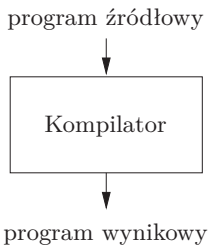
Książka ta jest poświęcona projektowaniu i implementacji kompilatorów. Pokażemy w niej, że kilka elementarnych koncepcji pozwala skonstruować translatory dla ogromnej różnorodności języków i maszyn. Oprócz samych kompilatorów, zasady i techniki ich projektowania mają zastosowanie w tak wielu innych dziedzinach, że zapewne zostaną wielokrotnie ponownie użyte w pracy informatyka. Studiowanie pisania kompilatorów oznacza poznawanie takich zagadnień, jak języki programowania, architektura komputerów, teoria języka, algorytmy i inżynieria oprogramowania.

W tym wstępnym rozdziale przedstawimy różne formy translatorów językowych, zaprezentujemy ogólny przegląd struktury typowego kompilatora i przedyskutujemy trendy występujące w językach programowania i architekturach komputerów, które mają wpływ na kształtowanie kompilatorów. Dołączymy też pewne spostrzeżenia dotyczące zależności między projektowaniem kompilatorów a teoriami informatycznymi oraz naszkicujemy zastosowania technik właściwych dla kompilacji, które wykraczają poza samą kompilację. Zakończymy skrótowym przedstawieniem kluczowych koncepcji języków programowania, które będą niezbędne w naszych badaniach nad kompilatorami.

1.1. Translatory

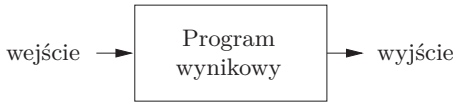
Mówiąc w uproszczeniu, kompilator jest programem, który potrafi przeczytać program sformułowany w jednym języku – języku *źródłowym* – i przełożyć

go na równoważny program w innym języku – języku *wynikowym* (patrz rysunek 1.1). Ważną rolą kompilatora jest zgłoszenie wykrytych w czasie tłumaczenia dowolnych błędów w programie źródłowym.



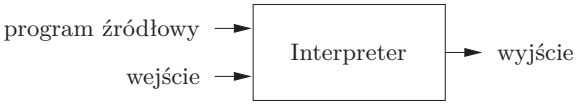
RYSUNEK 1.1: Kompilator

Jeśli program wynikowy jest programem wykonywalnym w języku maszynowym, może zostać uruchomiony przez użytkownika w celu przetworzeniu wejścia i wygenerowania wyjścia (patrz rysunek 1.2).



RYSUNEK 1.2: Uruchamianie wynikowego programu

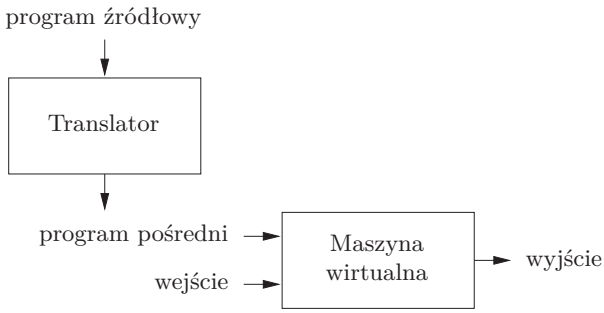
Innym powszechnie spotykanym rodzajem translatora jest *interpreter*. Zamiast tworzenia programu wynikowego jako efektu tłumaczenia, interpreter wydaje się bezpośrednio wykonywać operacje wyspecyfikowane w programie źródłowym względem danych wejściowych dostarczanych przez użytkownika, co demonstruje rysunek 1.3.



RYSUNEK 1.3: Interpreter

Z jednej strony program wynikowy w języku maszynowym tworzony przez kompilator jest zazwyczaj znacznie szybszy od interpretera przy przechodzeniu od wejścia do wyjścia. Z drugiej strony interpreter zazwyczaj udostępnia lepszą diagnostykę błędów niż kompilator, gdyż wykonuje program źródłowy instrukcja po instrukcji.

Przykład 1.1: Translatory języka Java łączą kompilację i interpretację, co pokazuje rysunek 1.4. Program źródłowy Java może zostać najpierw skompilowany do formy pośredniej nazywanej kodem bajtowym (*bytecode*). Kod bajtowy jest następnie interpretowany przez maszynę wirtualną. Zaletą tego podejścia jest to,

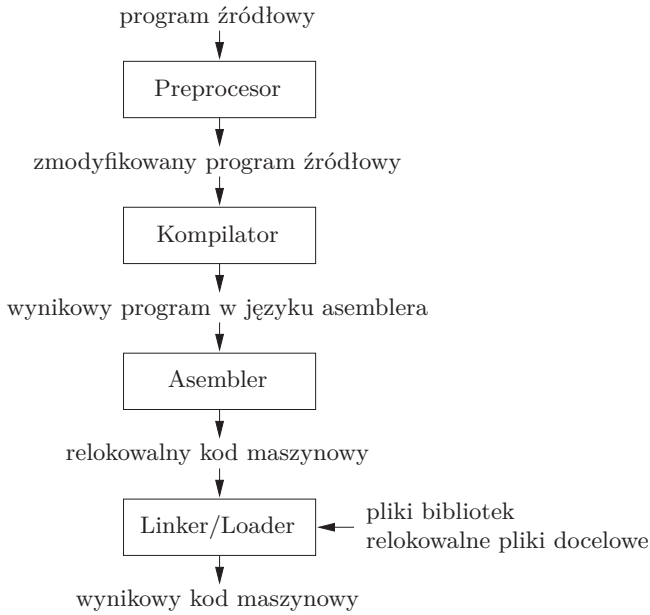


RYSUNEK 1.4: Kompilator hybrydowy

że kod bajtowy – skompilowany na jednej maszynie – może być interpretowany na innym komputerze, być może również przez sieć.

W dążeniu do uzyskania szybszego przetwarzania wejścia na wyjście niektóre kompilatory Java, nazywane kompilatorami *just-in-time* (JIT, kompilacja na żądanie), tłumaczą kod bajtowy na język maszynowy bezpośrednio przed uruchomieniem programu pośredniego w celu przetworzenia danych wejściowych. ■

Do utworzenia wynikowego programu wykonywalnego, oprócz kompilatora, może być potrzebnych kilka innych programów, co ilustruje rysunek 1.5. Program źródłowy może być podzielony na moduły przechowywane w oddzielnych plikach. Zadanie zbierania elementów programu źródłowego jest niekiedy powierzane



RYSUNEK 1.5: System przetwarzania języka (translacji)

oddzielnemu programowi, nazywanemu *preprocesorem*. Może on również rozwijać skróty nazywane makrami do pełnych wyrażeń języka źródłowego.

Zmodyfikowany program źródłowy jest następnie przekazywany do kompilatora. Ten może utworzyć jako swoje wyjście program w języku asemblera, gdyż tworzenie kodu asemblera jest łatwiejsze do wykonania, a ponadto łatwiejsze do debugowania. Uzyskany kod asemblera jest następnie przetwarzany przez kolejny program nazywany po prostu *asemblerem*, który generuje relokowalny kod maszynowy jako swoje wyjście.

Obszerne programy są często kompilowane w częściach, zatem uzyskane fragmenty kodu maszynowego mogą wymagać złączenia z innymi relokowalnymi plikami wynikowymi i plikami bibliotek w kod, który ostatecznie może zostać uruchomiony na komputerze. *Konsolidator* (nazywany również *linkerem*) rozwiązuje zewnętrzne adresy pamięci, dzięki czemu kod w jednym pliku może odnosić się do lokalizacji w innym pliku. Następnie program ładujący (*loader*) umieszcza wszystkie wykonywalne pliki obiektowe w pamięci w celu wykonania.

1.1.1. Ćwiczenia do podrozdziału 1.1

Ćwiczenie 1.1.1: Na czym polega różnica między kompilatorem a interpreterem?

Ćwiczenie 1.1.2: Jakie są zalety (a) kompilatora wobec interpretera oraz (b) interpretera wobec kompilatora?

Ćwiczenie 1.1.3: Jakie korzyści zapewnia system przetwarzania języka, w którym kompilator tworzy kod w języku asemblera, a nie w języku maszynowym?

Ćwiczenie 1.1.4: Kompilator tłumaczący jeden język wysokiego poziomu na inny język wysokiego poziomu jest nazywany translatorem *source-to-source* lub transkompilatorem. Jakie mogą być korzyści użycia języka C jako języka wynikowego kompilatora?

Ćwiczenie 1.1.5: Opisz kilka zadań, które musi wykonać asembler.

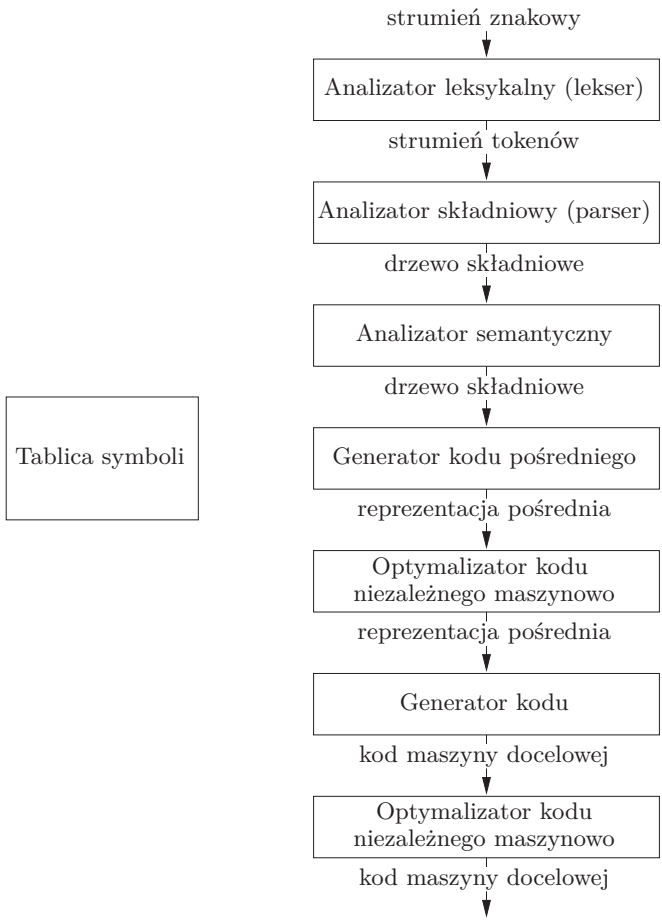
1.2. Struktura kompilatora

Do tego miejsca traktowaliśmy kompilator jako prostą skrzynkę, która mapuje program źródłowy na semantycznie równoważny program wynikowy. Jeśli jednak nieco uchylimy pokrywy tej skrzynki, zobaczymy, że mapowanie to jest wykonywane w dwóch fazach: analizy i syntezy.

Część analityczna dzieli program źródłowy na części składowe i stosuje do nich strukturę gramatyczną. Następnie używa tej struktury do utworzenia pośredniej reprezentacji programu źródłowego. Jeśli część analityczna odkryje, że program źródłowy jest błędnie sformułowany pod względem składniowym lub

niejednoznaczny semantycznie, musi zwrócić komunikaty informujące o tych niedociągnięciach, aby użytkownik mógł podjąć działania naprawcze. W części analitycznej gromadzone są również informacje o programie źródłowym, umieszczane w strukturze danych nazywanej tablicą symboli, która jest przekazywana wraz z reprezentacją pośrednią do fazy syntezy. Część syntezy konstruuje pożądany program wynikowy z reprezentacji pośredniej i informacji zawartych w tablicy symboli. Część analityczna kompilatora jest często określana terminem *front-end* lub po prostu przodem; część syntetyczna to *back-end*, czyli tył kompilatora.

Jeśli bardziej szczegółowo przyjrzymy się procesowi kompilacji, zobaczymy, że działa on jako sekwencja kilku faz, których każda przekształca jedną reprezentację programu źródłowego w kolejną. Typowy rozkład kompilatora na poszczególne fazy pokazuje rysunek 1.6. W praktyce wiele faz może być grupowanych



RYSUNEK 1.6: Fazy kompilacji

razem i reprezentacje pośrednie między zgrupowanymi fazami nie muszą być jawnie konstruowane. Tablica symboli, która przechowuje informacje o całym programie źródłowy, jest używana we wszystkich fazach działania kompilatora.

Niektóre kompilatory wykorzystują fazę optymalizacji niezależnej od maszyny między częścią przednią a tylną. Celem tej fazy optymalizacyjnej jest wykonanie przekształceń reprezentacji pośredniej, aby segment *back-end* mógł wytworzyć lepszy program wynikowy, niż powstały z niezoptymalizowanego kodu pośredniego. Jako że optymalizacja jest opcjonalna, jedna lub obie fazy optymalizacyjne widoczne na rysunku 1.6 mogą być nieobecne.

1.2.1. Analiza leksykalna

Pierwsza faza działania kompilatora nazywana jest analizą leksykalną lub skanowaniem. Analizator leksykalny (nazywany niekiedy *skanerem* lub *lekserem*) odczytuje strumień znaków budujących program źródłowy i grupuje te znaki w znaczące sekwencje nazywane *leksemami*. Dla każdego leksemu analizator leksykalny tworzy wyjście w postaci *tokenu* w formacie

(*nazwa-tokenu, wartość-atrybutu*)

który przekazywany jest do następnej fazy, czyli analizy składniowej. W tokenie pierwszy komponent *nazwa-tokenu* jest symbolem abstrakcyjnym, który używany jest podczas analizy składniowej, drugi zaś komponent *wartość-atrybutu* wskazuje wpis w tablicy symboli dla tego tokenu. Informacja zawarta we wpisie w tablicy symboli jest potrzebna dla analizy semantycznej i generowania kodu.

Na przykład przypuśćmy, że program źródłowy zawiera instrukcję przypisania

```
position = initial + rate * 60
```
(1.1)

Znaki w tym wyrażeniu mogą zostać pogrupowane w następujące leksemy i odwzorowane na następujące tokeny przekazywane do analizatora składniowego:

1. `position` jest leksemem, który zostanie „odwzorowany" na token ⟨**id**, 1⟩, gdzie **id** jest symbolem abstrakcyjnym oznaczającym *identyfikator*, 1 wskazuje zaś wpis w tablicy symboli dla `position`. Wpis w tabeli symboli dla identyfikatora przechowuje informacje o tym identyfikatorze, takie jak jego nazwa i typ.
2. Symbol przypisania = jest leksemem odwzorowanym na token ⟨=⟩. Ponieważ ten token nie potrzebuje wartości, pominęliśmy drugi komponent. Moglibyśmy użyć dowolnego abstrakcyjnego symbolu, takiego jak **assign** jako nazwy tokenu, ale dla wygody zapisu zdecydowaliśmy się użyć samego leksemu jako nazwy symbolu abstrakcyjnego.
3. `initial` jest leksemem odwzorowanym na ⟨**id**, 2⟩, przy czym 2 wskazuje wpis w tabeli symboli dla `initial`.
4. `+` jest leksemem odwzorowanym na token ⟨+⟩.

5. `rate` jest leksemem odwzorowanym na token ⟨**id**, 3⟩, gdzie 3 wskazuje wpis w tablicy symboli dla `rate`.
6. `*` jest leksemem odwzorowanym na token ⟨∗⟩.
7. `60` jest leksemem odwzorowanym na token ⟨60⟩[1].

Spacje oddzielające leksemy powinny zostać pominięte przez analizator leksykalne.

Rysunek 1.7 pokazuje reprezentację wyrażenia przypisania (1.1) po analizie leksykalnej jako sekwencję tokenów

$$\textbf{id}1\ \langle = \rangle\ \textbf{id}2\ \langle + \rangle\ \textbf{id}3\ \langle * \rangle\ \langle 60 \rangle \tag{1.2}$$

W tej reprezentacji nazwy tokenów =, + oraz * są symbolami abstrakcyjnymi dla operatorów przypisania, dodawania i mnożenia, odpowiednio.

1.2.2. Analiza składniowa

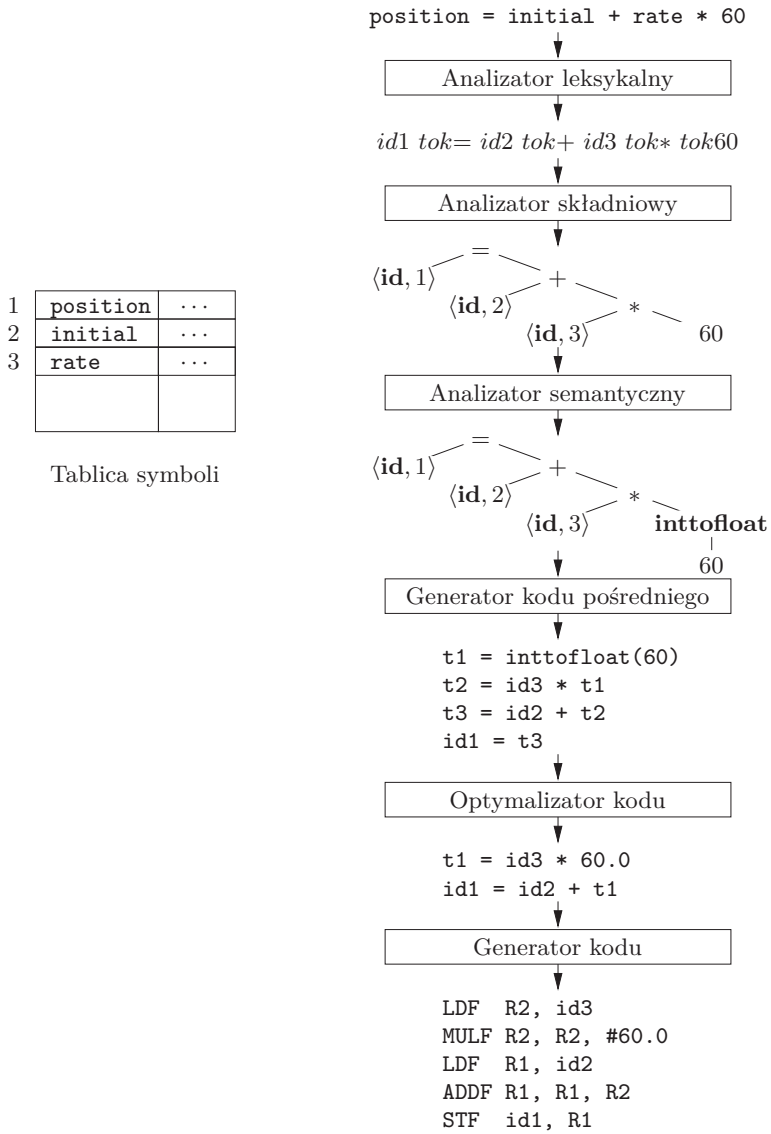
Druga faza pracy kompilatora to *analiza składniowa* (*parsing*). Analizator składniowy, nazywany też parserem, używa pierwszych komponentów tokenów utworzonych przez analizator leksykalny do zbudowania pośredniej reprezentacji przypominającej drzewo, odwzorowującej gramatyczną strukturę strumienia tokenów. Typową reprezentacją jest drzewo składniowe, w którym każdy wewnętrzny węzeł oznacza operację, a potomne gałęzie węzła reprezentują argumenty tej operacji. Drzewo składniowe dla strumienia tokenów (1.2) pokazuje rysunek 1.7 jako wyjście analizatora składniowego.

Drzewo to pokazuje kolejność, w jakiej wykonywane są operacje wchodzące w skład wyrażenia przypisania

```
position = initial + rate * 60
```

Drzewo to zawiera wewnętrzny węzeł z etykietą * dla ⟨**id**, 3⟩ jako lewego potomka i liczbę całkowitą 60 jako prawego potomka. Węzeł ⟨**id**, 3⟩ reprezentuje identyfikator `rate`. Węzeł z etykietą * jasno pokazuje, że musimy najpierw pomnożyć wartość `rate` przez 60. Węzeł oznaczony + pokazuje, że musimy dodać wynik tego mnożenia do wartości `initial`. Korzeń drzewa, oznaczony =, wskazuje, że musimy przechować wynik tego dodawania w lokalizacji wskazywanej przez identyfikator `position`. Takie uporządkowanie operacji jest spójne ze standardową konwencją arytmetyczną, głoszącą, że mnożenie ma wyższy priorytet niż dodawanie, a tym samym musi być wykonane przed dodawaniem.

[1] Z technicznego punktu widzenia dla leksemu `60` powinniśmy utworzyć token podobny do ⟨**number**, 4⟩, przy czym 4 wskazywałoby wpis w tablicy symboli dla wewnętrznej reprezentacji liczby całkowitej 60, ale na razie wolimy odłożyć omówienie tokenów dla liczb (stałych) do rozdziału 2. W rozdziale 3 omówimy techniki budowania analizatorów leksykalnych.



RYSUNEK 1.7: Translacja wyrażenia przypisania

Kolejne fazy działania kompilatora wykorzystują strukturę gramatyczną do analizowania programu źródłowego i wygenerowania programu wynikowego. W rozdziale 4 pokażemy użycie gramatyk bezkontekstowych do specyfikowania gramatycznej struktury języków programowania i omówimy algorytmy automatycznego konstruowania wydajnych analizatorów składniowych dla określonych klas gramatyk. W rozdziałach 2 i 5 pokażemy, że definicje sterowane składnią mogą pomóc w uściśleniu tłumaczenia konstrukcji języków programowania.

1.2.3. Analiza semantyczna

Analizator semantyczny wykorzystuje drzewo składniowe oraz informacje z tablicy symboli do sprawdzenia programu źródłowego pod kątem spójności semantycznej programu z definicją języka. Ponadto gromadzi on informacje o typach i zapisuje je albo w drzewie składniowym, albo w tablicy symboli do późniejszego użycia podczas generowania kodu pośredniego.

Ważną częścią analizy semantycznej jest *sprawdzanie typów*, w którym kompilator sprawdza, czy każdemu operatorowi odpowiadają pasujące operandy. Na przykład wiele definicji języków programowania wymaga, aby indeks tablicy był wartością całkowitą; kompilator musi zgłosić błąd, jeśli jako indeks tablicy zostanie użyta liczba zmiennoprzecinkowa.

Specyfikacja języka może pozwalać na konwersje pewnych typów nazywanych *koercjami* (konwersjami wymuszonymi). Na przykład binarny operator arytmetyczny może zostać zastosowany albo do pary liczb całkowitych, albo do pary liczb zmiennoprzecinkowych. Jeśli operator ten zostanie użyty wobec liczby zmiennoprzecinkowej i całkowitej, kompilator może przekonwertować liczbę całkowitą na zmiennoprzecinkową przed wykonaniem operacji.

Tego typu koercja jest widoczna na rysunku 1.7. Przypuśćmy, że position, initial oraz rate zostały zadeklarowane jako liczby zmiennoprzecinkowe, a leksem 60 sam z siebie tworzy liczbę całkowitą. Element sprawdzania typu w analizatorze semantycznym na rysunku 1.7 odkrywa, że operator * jest zastosowany do zmiennoprzecinkowej liczby rate oraz całkowitej stałej 60. W tym przypadku liczba całkowita może zostać przekonwertowana na liczbę zmiennoprzecinkową. Na rysunku 1.7 można zauważyć, że wyjście analizatora semantycznego ma dodatkowy węzeł dla operatora **inttofloat**, który jawnie konwertuje swój argument całkowitoliczbowy na liczbę zmiennoprzecinkową. Sprawdzanie typów i analiza semantyczna są omówione w rozdziale 6.

1.2.4. Generowanie kodu pośredniego

W procesie tłumaczenia programu źródłowego na kod wynikowy kompilator może konstruować jedną lub więcej pośrednich reprezentacji, które mogą mieć rozmaite formy. Drzewa składniowe są jedną z wielu postaci reprezentacji pośrednich; są one typowo używane w analizie składniowej i semantycznej.

Po analizie składniowej i semantycznej programu źródłowego wiele kompilatorów generuje jawną niskopoziomową reprezentację pośrednią, zbliżoną do kodu maszynowego, o której możemy myśleć jako o programie dla maszyny abstrakcyjnej. Ta postać pośrednia powinna mieć dwie ważne właściwości: powinna być łatwa do utworzenia i łatwa do przetłumaczenia na kod maszyny docelowej.

W rozdziale 6 rozważymy formę pośrednią nazywaną kodem trójadresowym (*three-address code*, TAC), składającą się z sekwencji instrukcji przypominających asembler, przy czym każda instrukcja ma co najwyżej trzy operandy. Każdy operand może działać jak rejestr. Wyjście generatora kodu pośredniego widoczne

na rysunku 1.7 składa się z sekwencji kodu trójadresowego

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

Istnieje kilka zagadnień dotyczących instrukcji trójadresowych, o których warto wspomnieć. Po pierwsze, każda trójadresowa instrukcja przypisania może zawierać co najwyżej jeden operator po prawej stronie. Tym samym instrukcje te ustalają kolejność, w której operacje mają być wykonane; mnożenie poprzedza dodawanie, choć w źródłowym programie zostało umieszczone później (1.1). Po drugie, kompilator musi wygenerować tymczasowe nazwy do przechowania wartości obliczanej przez instrukcję trójadresową. Po trzecie, niektóre „trójadresowe instrukcje", jak pierwsza i ostatnia w powyższej sekwencji (1.3), mają mniej niż trzy operandy.

W rozdziale 6 przedstawimy główne reprezentacje pośrednie używane w kompilatorach. W rozdziale 5 wprowadzimy techniki tłumaczenia sterowane składnią, które zostaną zastosowane w rozdziale 6 do sprawdzania typów i generowania kodu pośredniego dla typowych konstrukcji języków programowania, takich jak wyrażenia, instrukcje sterowania przepływem i wywołania procedur.

1.2.5. Optymalizacja kodu

Faza niezależnej od architektury maszynowej optymalizacji kodu ma na celu ulepszenie kodu pośredniego, dzięki czemu lepszy będzie również kod wynikowy. Zazwyczaj „lepszy" oznacza „szybszy", ale możliwe są też inne pożądane cele, takie jak mniejsza długość lub mniejsze zużycie energii przez kod wynikowy. Na przykład bezpośredni (niezoptymalizowany) algorytm generuje kod pośredni (1.3), używając pojedynczej instrukcji dla każdego operatora występującego w reprezentacji drzewa pochodzącej z analizatora semantycznego.

Użycie prostego algorytmu do generowania kodu pośredniego, po którym następuje faza optymalizacji, jest rozsądną metodą wygenerowania dobrego kodu wynikowego. Optymalizator może wydedukować, że konwersja liczby 60 z typu całkowitoliczbowego na zmiennoprzecinkowy może zostać wykonana raz na zawsze w czasie kompilacji, zatem operację **inttofloat** można wyeliminować, zastępując całkowitą stałą 60 liczbą zmiennoprzecinkową 60.0. Co więcej, `t3` jest używana tylko raz w celu przekazania swojej wartości do id1, zatem optymalizator może przekształcić sekwencję (1.3) w krótszą:

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

Istnieje wielkie zróżnicowanie tego, jak wiele optymalizacji kodu wykonują różne kompilatory. W przypadku tych, które robią najwięcej optymalizacji – tak zwanych „optymalizujących kompilatorów" – faza ta zajmuje istotną ilość czasu.

Istnieją też proste optymalizacje, które znacząco poprawiają czas działania wynikowego programu bez nadmiernego spowolniania kompilacji. Szczegółami optymalizacji, zarówno niezależnej, jak i zależnej od architektury maszynowej, zajmiemy się później, począwszy od rozdziału 8.

1.2.6. Generowanie kodu

Generator kodu przyjmuje jako wejście reprezentację pośrednią programu źródłowego i odwzorowuje ją na język wynikowy. Jeśli językiem tym jest kod maszynowy, dla każdej zmiennej używanej przez program wybierane są rejestry lub lokalizacje w pamięci. Następnie instrukcje pośrednie są tłumaczone na sekwencje instrukcji maszynowych wykonujących to samo zadanie. Krytycznym aspektem generowania kodu jest rozważne przypisanie rejestrów do przechowywanych zmiennych.

Na przykład przy używaniu rejestrów `R1` i `R2` kod pośredni (1.4) może zostać przetłumaczony na kod maszynowy

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2                    (1.5)
ADDF R1,  R1, R2
STF  id1, R1
```

Pierwszy operand każdej instrukcji wskazuje miejsce docelowe. Litera `F` w każdej instrukcji mówi nam, że dotyczą one liczb zmiennoprzecinkowych. Kod pokazany w wydruku (1.5) ładuje zawartość adresu `id3` do rejestru `R2`, po czym mnoży ją przez zmiennoprzecinkową stałą 60.0. Symbol `#` wskazuje, że 60.0 ma być traktowana jako natychmiastowa stała. Trzecia instrukcja przenosi `id2` do rejestru `R1`, czwarta zaś dodaje do niej wcześniej obliczoną wartość zapisaną w rejestrze `R2`. Na koniec wartość z rejestru `R1` zostaje zapisana pod adresem id1, tak więc kod poprawnie implementuje instrukcję przypisania (1.1). Generowanie kodu zostanie omówione w rozdziale 8.

To omówienie generowania kodu pomija ważny problem alokowania pamięci dla identyfikatorów w programie źródłowym. Jak zobaczymy w rozdziale 7, organizacja pamięci w czasie wykonania zależy od kompilowanego języka. Decyzje o alokacji pamięci są podejmowane podczas generowania kodu pośredniego lub w trakcie generowania kodu maszynowego (wynikowego).

1.2.7. Zarządzanie tablicą symboli

Kluczową funkcją kompilatora jest zarejestrowanie nazw zmiennych używanych w programie źródłowym i zebranie informacji o różnych atrybutach każdej nazwy. Atrybuty te mogą udostępniać informacje o pamięci alokowanej dla danej nazwy, jej typie, zakresie widzialności (gdzie w programie może być użyta wartość tej zmiennej). W przypadku nazw procedur rejestrowane są

takie elementy, jak liczba i typy jej argumentów, metoda przekazania każdego argumentu (na przykład przez wartość lub przez referencję) oraz zwracany typ.

Tablica symboli jest strukturą danych zawierającą rekord dla wszystkich nazw zmiennych, z polami dla atrybutów tych nazw. Ta struktura danych powinna zostać tak zaprojektowana, aby umożliwić kompilatorowi szybkie odszukanie rekordu dla każdej nazwy, jak również szybkie zapisywanie i odczytywanie danych z tego rekordu. Tablice symboli zostaną omówione w rozdziale 2.

1.2.8. Grupowanie faz w przebiegi

Omawiane fazy kompilacji odnoszą się do logicznej organizacji kompilatora. W konkretnej implementacji aktywność z wielu faz może zostać pogrupowana łącznie w przebieg (*pass*) odczytujący plik wejściowy i zapisujący plik wyjściowy. Na przykład fazy analizy leksykalnej, składniowej i semantycznej wraz z generowaniem kodu pośredniego mogą zostać zgrupowane w jeden przebieg. Optymalizacja kodu może być opcjonalnym przebiegiem. Na koniec możemy mieć przebieg części back-endu, wykonujący generowanie kodu dla określonej maszyny docelowej.

Niektóre zestawy (kolekcje) kompilatorów zostały zbudowane wokół starannie zaprojektowanych reprezentacji pośrednich, które pozwalają na połączenie przodu dla określonego języka z tyłem dla wybranej maszyny docelowej. W tych zestawach można tworzyć kompilatory dla różnych języków źródłowych dla jednej maszyny docelowej przez dołączanie różnych front-endów do back-endu dla tej maszyny.

Analogicznie, możemy łatwo utworzyć kompilatory dla różnych architektur docelowych maszyn, łącząc front-end dla wybranego języka z back-endami dla różnych maszyn.

1.2.9. Narzędzia do budowania kompilatorów

Twórca kompilatorów, podobnie jak każdy inny programista, może z powodzeniem wykorzystywać nowoczesne środowiska projektowania oprogramowania zawierające takie narzędzia, jak edytory dla poszczególnych języków, debugery, mechanizmy zarządzania wersjami, profilery, jarzma testowe i tak dalej. Oprócz tych ogólnych narzędzi programistycznych istnieją również bardziej specjalistyczne narzędzia, które powstały w celu ułatwienia implementacji różnych faz kompilatora.

Narzędzia te używają specjalizowanych języków do specyfikowania i implementowania określonych komponentów i mogą używać naprawdę wyrafinowanych algorytmów. Większość udanych narzędzi to te, które ukrywają szczegóły algorytmu generującego i tworzą komponenty, które mogą być łatwo zintegrowane z pozostałymi częściami kompilatora. Do często używanych narzędzi konstruujących kompilatory należą:

1. Generatory parserów, które automatycznie tworzą moduły analizy składniowej na podstawie gramatycznego opisu języka programowania.
2. Generatory lekserów, które tworzą analizatory leksykalne na podstawie opisu tokenów języka w postaci wyrażeń regularnych.
3. Sterowane składnią mechanizmy translacyjne, które tworzą zestawy procedur pozwalających na przechodzenie przez drzewo i generowanie pośredniego kodu.
4. Generatory generatorów kodu, które tworzą generatory kodu na podstawie zbioru reguł opisujących tłumaczenie każdej operacji języka pośredniego na język maszynowy docelowej architektury.
5. Silniki przepływu danych, które wykorzystują informacje o tym, jak wartości są przekazywane z jednej części programu do innych części. Analiza przepływu danych jest kluczową częścią optymalizacji kodu.
6. Przyborniki budowania kompilatorów, które udostępniają zintegrowane zestawy procedur dla budowania różnych faz kompilatora.

Wiele z tych narzędzi opiszemy w kolejnych częściach tej książki.

1.3. Ewolucja języków programowania

Pierwsze komputery elektroniczne pojawiły się w latach 40. XX w. i były programowane przy użyciu języka maszynowego – przez sekwencje zer i jedynek, które jawnie nakazywały komputerowi, co ma zrobić i w jakiej kolejności. Same operacje były bardzo niskiego poziomu: przenieś dane z tej lokalizacji do innej, dodaj zawartość dwóch rejestrów, porównaj dwie wartości i tak dalej. Nie musimy dodawać, że tego rodzaju programowanie było powolne, mozolne i podatne na błędy. A gdy już zostały napisane, programy były trudne do zrozumienia i modyfikowania.

1.3.1. Przejście na języki wyższego poziomu

Pierwszym krokiem w kierunku programowania bardziej przyjaznego dla człowieka stało się wynalezienie mnemonicznych języków asemblerów na początku lat 50. Początkowo instrukcje w języku asemblera były po prostu mnemonicznymi reprezentacjami instrukcji maszynowych. Później do języków asemblerów dodano makra – instrukcje, które mogły definiować sparametryzowane skróty dla często używanych sekwencji instrukcji maszynowych.

Wielki krok w kierunku języków wyższego poziomu dokonał się w drugiej połowie lat 50. XX w.: zaprojektowanie języka Fortran na potrzeby obliczeń naukowych, Cobol do przetwarzania danych biznesowych i Lisp do obliczeń symbolicznych. Filozofia leżąca w tle tych języków sprowadzała się do utworzenia

zapisów wyższego poziomu, dzięki którym programiści mogli łatwiej pisać obliczenia numeryczne, aplikacje biznesowe i programy operujące na symbolach. Języki te były tak udane, że dziś nadal są używane.

W kolejnych dekadach powstało wiele nowych języków zawierających innowacyjne cechy, które pomagały sprawić, aby programowanie stało się łatwiejsze, bardziej naturalne, a przede wszystkim niezawodne. W dalszej części tego rozdziału omówimy kilka kluczowych właściwości, które są wspólne dla wielu współczesnych języków programowania.

Obecnie mamy tysiące języków programowania. Można je klasyfikować na wiele sposobów. Jedną z tych klasyfikacji jest pojęcie generacji. Języki *pierwszej generacji* to języki maszynowe, *druga generacja* to asemblery, trzecia zaś to języki wyższego poziomu, takie jak Fortran, Cobol, Lisp, C, C++, C# czy Java. Języki *czwartej generacji* to języki zaprojektowane do określonych zastosowań, na przykład NOMAD do generowania raportów, SQL dla zapytań do baz danych lub Postscript do formatowania tekstu. Termin *języki piątej generacji* jest stosowany do języków opartych na logice lub ograniczeniach takich jak Prolog i OPS5.

Inna klasyfikacja języków używa pojęcia *imperatywne* dla języków, w których program określa, *jak* obliczenia mają być wykonywane, i *deklaratywne* dla języków, w których program określa, *jakie* obliczenia mają być wykonane. Języki, takie jak C, C++, C# i Java, są językami imperatywnymi. W językach imperatywnych istnieje pojęcie stanu programu oraz instrukcji, które zmieniają ten stan. Języki funkcyjne, takie jak ML czy Haskell, a także języki z ograniczeniami logicznymi, takie jak Prolog, są zwykle postrzegane jako języki deklaratywne.

Termin *język von Neumanna* dotyczy języków programowania, których model obliczeniowy jest oparty na architekturze komputera zdefiniowanej przez von Neumanna. Wiele współczesnych języków, takich jak Fortran i C, są językami von Neumanna.

Język *zorientowany obiektowo* to taki, który obsługuje programowanie obiektowe, czyli styl programowania, w którym program składa się ze zbioru obiektów, które wchodzą ze sobą w interakcje. Simula 67 oraz Smalltalk były pierwszymi ważnymi językami zorientowanymi obiektowo. Przykłady bardziej współczesnych języków obiektowych to C++, C#, Java oraz Ruby.

Języki skryptowe to języki interpretowane wyposażone w operatory wysokiego poziomu, zaprojektowane do wykonywania obliczeń „sklejających całość". Te obliczenia były pierwotnie nazywane „skryptami". Awk, JavaScript, Perl, PHP, Python, Ruby i Tcl to przykłady popularnych języków skryptowych. Programy pisane w językach skryptowych są często znacznie krótsze od równoważnych programów napisanych w takich językach, jak C.

1.3.2. Wpływ na kompilatory

Jako że projektowanie języków programowania i kompilatorów jest nierozerwalnie związane, postępy w językach programowania stawiały nowe wyzwania przed autorami kompilatorów. Musieli oni wymyślać algorytmy i reprezentacje do

tłumaczenia oraz zapewnić obsługę nowych funkcji języków. Od lat 40. XX w. ewoluowała również architektura komputerów. Twórcy kompilatorów musieli nie tylko śledzić nowe funkcje języków, lecz także wynajdywać algorytmy tłumaczące, które zapewniłyby maksymalne wykorzystanie nowych możliwości sprzętowych.

Kompilatory mogą wspomóc promowanie użycia języków wysokiego poziomu, minimalizując narzut programów pisanych w tych językach. Kompilatory odgrywają też krytyczną rolę w zapewnianiu faktycznego wykorzystania wysokowydajnej architektury komputerów w aplikacjach użytkowników. W istocie wydajność systemu komputerowego jest tak uzależniona od technologii kompilatorów, że są one używane jako narzędzie oceniania koncepcji architektonicznych, zanim komputer zostanie faktycznie zbudowany.

Tworzenie kompilatorów jest wymagającym zadaniem. Kompilator sam z siebie jest dużym programem. Co więcej, wiele nowoczesnych systemów przetwarzania języków obsługuje wiele źródłowych języków i maszyn docelowych w obrębie tej samej platformy; innymi słowy, systemy te służą jako zbiory kompilatorów obejmujących łącznie miliony wierszy kodu. W konsekwencji przy tworzeniu i modernizowaniu nowoczesnych procesorów języków kluczowe jest stosowanie dobrych technik projektowania oprogramowania.

Kompilator musi poprawnie przetłumaczyć potencjalnie nieskończony zbiór programów, które można napisać w języku źródłowym. Problem generowania optymalnego kodu wynikowego z programu źródłowego jest w ogólności nierozstrzygalny; tym samym twórcy kompilatorów muszą dokonywać kompromisów i ostrożnych wyborów, jakimi problemami mają się zajmować i jakiej heurystyki użyć w celu rozwiązania problemu generowania wydajnego kodu.

Studiowanie kompilatorów jest też analizowaniem tego, jak teoria spotyka się z praktyką, co zobaczymy w podrozdziale 1.4.

Podręcznik ten ma na celu nauczenie metodologii i podstawowych idei używanych w projektowaniu kompilatorów. Nie jest naszą intencją przedstawienie wszystkich algorytmów i technik, które mogłyby zostać użyte do budowania perfekcyjnego systemu przetwarzania języków. Jednak czytelnicy uzyskają podstawową wiedzę i zrozumienie pojęć, które pozwolą im względnie łatwo nauczyć się budowania kompilatorów.

1.3.3. Ćwiczenia do podrozdziału 1.3

Ćwiczenie 1.3.1: Określ, które z poniższych terminów:

(a) imperatywny	(b) deklaratywny	(c) von Neumanna
(d) zorientowany obiektowo	(e) funkcyjny	(f) trzeciej generacji
(g) czwartej generacji	(h) skryptowy	

stosuje się do następujących języków:

(1) C	(2) C++	(3) Cobol	(4) Fortran	(5) Java
(6) Lisp	(7) ML	(8) Perl	(9) Python	(10) VB.

1.4. Teoria konstruowania kompilatorów

Projekty kompilatorów pełne są pięknych przykładów, w których złożone problemy praktyczne są rozwiązywane przez matematyczną abstrakcję istoty problemu. Stanowią one doskonałe ilustracje tego, jak można użyć abstrakcji do rozwiązywania problemów: bierzemy problem, formułujemy matematyczne uogólnienie uwzględniające jego kluczowe cechy i rozwiązujemy go przy użyciu technik matematycznych. Sformułowanie problemu musi opierać się na dobrym rozumieniu cech charakterystycznych programów komputerowych, rozwiązanie zaś musi zostać empirycznie zweryfikowane i dostrojone.

Kompilator musi akceptować wszystkie programy źródłowe zgodne ze specyfikacją języka; zbiór programów źródłowych jest zasadniczo nieskończony, a dowolny program może być bardzo obszerny i składać się potencjalnie z milionów wierszy kodu. Dowolne transformacje wykonywane przez kompilator podczas tłumaczenia programu źródłowego muszą zachowywać sens kompilowanego programu. Twórcy kompilatorów mają zatem wpływ nie tylko na pisane przez siebie kompilatory, lecz także na wszelkie programy generowane przez te kompilatory. To uwarunkowanie sprawia, że pisanie kompilatorów jest szczególnie satysfakcjonujące; jednak sprawia też, że projektowanie kompilatorów jest prawdziwym wyzwaniem.

1.4.1. Modelowanie w projektowaniu i implementacji kompilatora

Badanie kompilatorów polega głównie na studiowaniu, jak projektować właściwe modele matematyczne i wybierać odpowiednie algorytmy, zapewniając równowagę między wymaganiem ogólności i skuteczności a prostotą i wydajnością.

Niektóre z najbardziej podstawowych modeli to automaty skończone (*finite-state machine*, FSM) i wyrażenia regularne, którymi zajmiemy się w rozdziale 3. Modele te są przydatne przy opisywaniu jednostek leksykalnych programów (słów kluczowych, identyfikatorów i temu podobnych) i opisywaniu algorytmów używanych przez kompilator w celu rozpoznawania tych jednostek. Do fundamentalnych modeli należą również gramatyki bezkontekstowe służące do opisywania struktur składniowych języków programowania, takich jak zagnieżdżanie nawiasów lub konstrukcje sterujące. Gramatykami zajmiemy się w rozdziale 4. Innym ważnym modelem reprezentującym strukturę programu i ich translację na kod obiektowy są drzewa, co zobaczymy w rozdziale 5.

1.4.2. Nauka o optymalizacji kodu

Termin „optymalizacja” w kontekście projektowania kompilatorów odnosi się do prób podejmowanych przez kompilator w celu wytworzenia kodu bardziej wydajnego niż kod oczywisty (narzucający się). „Optymalizacja” jest więc błędną

nazwą, jako że nie ma możliwości zagwarantowania, że kod utworzony przez kompilator będzie rzeczywiście równie szybki lub szybszy niż dowolny inny kod realizujący to samo zadanie.

Współcześnie optymalizacja kodu wykonywana przez kompilator stała się jednocześnie bardziej ważna, ale i bardziej złożona. Większa złożoność wynika stąd, że architektury procesorów również stały się bardziej rozbudowane, zapewniając więcej możliwości usprawnienia sposobu wykonywania kodu. Jest bardziej ważna, gdyż masywnie równoległe komputery wymagają znaczących optymalizacji; w przeciwnym razie ich wydajność może spaść o rząd wielkości lub jeszcze bardziej. Wraz z dominacją maszyn wielordzeniowych (komputerów wyposażonych w układy zawierające wielką liczbę procesorów) wszystkie kompilatory muszą poradzić sobie z problemem rzeczywistego wykorzystania maszyn wieloprocesorowych.

Byłoby trudne, jeśli nie niemożliwe, zbudowanie niezawodnego kompilatora z gotowych „sztuczek". Z tego względu wokół problemu optymalizacji kodu powstała rozległa i użyteczna teoria. Wykorzystanie ścisłych podstaw matematycznych pozwala dowieść, czy optymalizacja jest poprawna i czy daje pożądany efekt dla wszelkich możliwych danych wejściowych. Począwszy od rozdziału 9, pokażemy, że takie modele, jak grafy, macierze i programowanie liniowe są niezbędne, jeśli kompilator ma tworzyć dobrze zoptymalizowany kod.

Jednocześnie sama czysta teoria nie jest wystarczająca. Podobnie jak w przypadku wielu innych praktycznych problemów, nie istnieją ostateczne i doskonałe odpowiedzi. W istocie większość pytań, które stawiamy w związku z optymalizacją kompilatorów, jest nierozstrzygalna. Jedną z najważniejszych umiejętności potrzebnych w projektowaniu kompilatorów jest zdolność do właściwego sformułowania problemu, który chcemy rozwiązać. Na początek potrzebujemy dobrego rozumienia zachowania programów oraz pogłębionego eksperymentowania i testowania w celu weryfikacji naszych intuicji.

Optymalizacja kompilatora musi spełniać następujące cele projektowe:

- Optymalizacja musi być poprawna, czyli zachowywać znaczenie skompilowanego programu.
- Optymalizacja musi poprawiać wydajność wielu programów.
- Czas kompilacji musi nadal być rozsądny.
- Wymagany wysiłek projektowy musi być akceptowalny.

Niemożliwe byłoby przecenienie ważności poprawności optymalizacji. Trywialne byłoby napisanie kompilatora generującego szybki kod, gdyby ten wygenerowany kod nie musiał być poprawny! Optymalizowanie kompilatorów jest tak trudne, że możemy z przekonaniem powiedzieć, że nie istnieje optymalizujący kompilator całkowicie wolny od błędów!

Tym samym najważniejszym celem, który trzeba mieć na względzie przy pisaniu kompilatora, jest poprawność.

Drugi cel głosi, że kompilator musi być skuteczny w poprawianiu wydajności wielu wejściowych programów. Na ogół przez wydajność rozumiemy

szybkość wykonywania programu. Szczególnie w przypadku wbudowanych aplikacji możemy również dążyć do minimalizowania wielkości wygenerowanego kodu. W przypadku zaś urządzeń przenośnych pożądane byłoby również, aby kod ograniczał zużycie energii. Zazwyczaj te same optymalizacje, które skracają czas wykonania, również oszczędzają energię. Poza wydajnością ważne mogą być również aspekty użytkowalności, takie jak zgłaszanie błędów i debugowanie.

Ponadto musimy utrzymać dostatecznie krótki czas kompilacji, aby zapewnić szybki cykl programowania i debugowania. Wymóg ten jest obecnie łatwiejszy do spełnienia w miarę, jak komputery są coraz szybsze. Często program jest najpierw pisany i debugowany bez optymalizacji. Nie tylko pozwala to na redukcję czasu kompilacji, lecz także, co ważniejsze, niezoptymalizowane programy są łatwiejsze do debugowania, gdyż usprawnienia wprowadzane przez kompilator często zaciemniają powiązanie między kodem źródłowym a wynikowym. Włączenie optymalizacji w kompilatorze niekiedy ujawnia nowe problemy w programie źródłowym; tym samym konieczne jest ponowne przeprowadzenie testów zoptymalizowanego kodu. Ta potrzeba dodatkowych testów niekiedy odstręcza od użycia optymalizacji w aplikacjach, szczególnie wtedy, gdy ich wydajność nie jest krytyczna.

Na koniec, kompilator jest złożonym systemem; musimy zadbać o to, aby był to system możliwie prosty, aby zagwarantować, że koszty jego budowy i utrzymania były możliwe do przyjęcia. Istnieje praktycznie nieskończona liczba optymalizacji, które moglibyśmy zaimplementować, a stworzenie poprawnych i skutecznych optymalizacji wymaga nietrywialnego wysiłku. Konieczne jest określenie priorytetów i implementowanie tylko tych optymalizacji, które prowadzą do największych korzyści w praktycznie spotykanych programach źródłowych.

Przy studiowaniu kompilatorów uczymy się zatem nie tylko, jak je budować, lecz także ogólnej metodologii rozwiązywania złożonych i otwartych problemów. Podejście używane w projektowaniu kompilatorów obejmuje zarówno teorie, jak i eksperymenty. Zazwyczaj rozpoczynamy od sformułowania problemu na podstawie intuicyjnego założenia, jakie trudności są istotne.

1.5. Zastosowania technologii kompilatorów

Projektowanie kompilatorów nie dotyczy jedynie kompilatorów jako takich. Wiele osób używa technik poznanych podczas studiowania kompilatorów, choć nigdy, mówiąc ściśle, nie napisali (nawet części) kompilatora dla któregoś z głównych języków programowania. Technologia kompilatorów ma również inne ważne zastosowania. Dodatkowo projekty kompilatorów mają wpływ na wiele innych obszarów informatyki. W tym podrozdziale przejrzymy najważniejsze zależności i zastosowania tej technologii.

1.5.1. Implementacja języków programowania wysokiego poziomu

Język programowania wysokiego poziomu definiuje poziom abstrakcji: programista formułuje algorytm przy użyciu tego języka, kompilator zaś musi przetłumaczyć ten program na język docelowy. W ogólności języki programowania wyższego poziomu ułatwiają programowanie, ale są mniej wydajne, co należy rozumieć, że wynikowe programy działają wolniej. Programiści używający języków niższego poziomu mają większą kontrolę nad przetwarzaniem i mogą z zasady tworzyć bardziej wydajny kod.

Nieszczęśliwie, programy tworzone w językach niższych poziomów są trudniejsze w pisaniu i – co jeszcze gorzej – mniej przenośne, bardziej podatne na błędy i trudniejsze do utrzymania. Optymalizujące kompilatory zawierają techniki poprawiające wydajność generowanego kodu, tym samym marginalizując nieefektywność wynikającą z abstrakcji wyższego poziomu.

Przykład 1.2: Słowo kluczowe **register** w języku C jest przykładem interakcji między technologią kompilatorów a ewolucją języka. Gdy język C powstawał w połowie lat 70., za konieczne uważano umożliwienie zapewnienia programiście kontroli nad tym, które zmienne programu mają być przechowywane w rejestrach. Ten poziom kontroli stał się niepotrzebny wraz z wynalezieniem skutecznych technik alokowania rejestrów i większość nowoczesnych programów nie używa już tej funkcjonalności języka.

W istocie programy używające słowa kluczowego **register** mogą tracić na wydajności, gdyż programiści często nie są najlepszymi decydentami w zagadnieniach bardzo niskiego poziomu, takich jak alokowanie rejestrów. Optymalny wybór alokacji zależy w największym stopniu od specyfiki architektury maszyny. Wpisanie na sztywno decyzji o zarządzaniu niskopoziomowymi zasobami, takimi jak rejestry, może w rezultacie bardzo zaszkodzić wydajności, szczególnie gdy program zostanie uruchomiony na innym komputerze niż ten, dla którego go napisano. ■

Wiele zmian w chętnie wybieranych językach programowania odbywało się w kierunku zwiększenia poziom abstrakcji. C był dominującym językiem programowania systemowego w latach 80.; wiele nowych projektów rozpoczynanych w latach 90. wykorzystywało C++; język Java, wprowadzony w roku 1995, szybko zdobył popularność w późnych latach 90. Nowe funkcjonalności kolejno wprowadzanych języków programowania pobudzały nowe badania w kierunku optymalizacji kompilatorów. W następnych podrozdziałach przedstawimy główne cechy języków, które stymulowały znaczący postęp w technologiach kompilatorów.

Praktycznie wszystkie popularne języki programowania, w tym C, Fortran i Cobol, obsługują złożone typy danych definiowanych przez użytkownika, takie jak tablice i struktury, oraz elementy kontroli przebiegu programu wyższego poziomu, takie jak pętle i wywoływanie procedur. Gdybyśmy po prostu wzięli każdą konstrukcję wysokiego poziomu lub operację dostępu do danych oraz przełożyli je wprost na kod maszynowy, wynik byłby bardzo nieefektywny. Podstawowy

element optymalizacji na poziomie kompilatora, znany jako *optymalizacja przepływu danych*, został opracowany w celu analizy przepływu danych przez program i usunięcia nadmiarowości występujących w tych konstrukcjach. Zaowocowało to generowaniem kodu, który przypomina kod napisany przez wykwalifikowanego programistę na niższym poziomie.

Podejście obiektowe zostało wprowadzone po raz pierwszy w języku Simula w roku 1967, a później zostało włączone do takich języków, jak Smalltalk, C++, C# i Java. Kluczowe koncepcje podejścia obiektowego to:

1. Abstrakcja danych.
2. Dziedziczenie właściwości.

Przy czym obie koncepcje pozwalają tworzyć programy bardziej modularne i łatwiejsze w utrzymaniu. Programy zorientowane obiektowo różnią się od tych pisanych w wielu innych językach pod tym względem, że składają się ze znacznie liczniejszych, ale mniejszych procedur (w terminologii obiektowej nazywanych *metodami*). Tym samym optymalizacje kompilatora muszą być w stanie działać dobrze ponad granicami, które stwarza struktura procedur programu źródłowego. Wstawianie kodu procedur, polegające na zastąpieniu wywołania procedury samym ciałem tej procedury, jest tu szczególnie użyteczne. Zostały również opracowane optymalizacje przyśpieszające rozgłaszanie metod wirtualnych.

Java zawiera liczne funkcjonalności ułatwiające pracę programisty, przy czym wiele z nich zostało wcześniej wprowadzonych w innych językach. Język Java jest bezpieczny typowo; innymi słowy, nie można użyć obiektu jako obiektu o nieprzypisanym typie. Wszelkie odwołania do tablicy są sprawdzane w celu upewnienia się, że leżą one wewnątrz granic tej tablicy. Java nie używa wskaźników i nie umożliwia arytmetyki na wskaźnikach. Zawiera wbudowany mechanizm odśmiecania pamięci automatycznie zwalniający pamięć zajmowaną przez zmienne, które nie są już używane. Choć wszystkie te cechy sprawiają, że programowanie jest łatwiejsze, powodują one dodatkowe obciążenie w czasie wykonywania programu. Optymalizacje kompilatorów zostały opracowane po to, aby zmniejszyć to obciążenie, przykładowo eliminując niepotrzebne sprawdzanie zakresów i umieszczanie na stosie, a nie na stercie obiektów, które nie muszą być dostępne spoza procedury. Zostały też opracowane efektywne algorytmy minimalizujące obciążenie działaniem mechanizmu odśmiecania pamięci.

Dodatkowo język Java został zaprojektowany do obsługi kodu przenośnego i mobilnego. Programy są rozpowszechniane w formie kodu bajtowego Java, który musi być albo interpretowany, albo skompilowany do kodu natywnego dynamicznie, czyli podczas uruchamiania. Dynamiczna kompilacja jest również rozważana w innych kontekstach, gdy informacje są wydobywane dynamicznie podczas uruchamiania i wykorzystywane do wytworzenia lepiej zoptymalizowanego kodu. W optymalizacji dynamicznej ważną kwestią jest minimalizacja czasu kompilacji, jako że stanowi ona część obciążenia wykonawczego. Typową techniką używaną w tym celu jest kompilowanie i optymalizowanie tylko tych części programu, które będą często wykonywane.

1.5.2. Optymalizacje architektur komputerów

Błyskawiczna ewolucja architektur komputerów również doprowadziła do niedającego się zaspokoić zapotrzebowania na nowe technologie kompilatorów. Niemal wszystkie wysoko wydajne systemy wykorzystują te same dwie podstawowe techniki: równoległość i hierarchie pamięci. Równoległość możemy zauważyć na wielu poziomach: na poziomie instrukcji, gdy wiele operacji jest wykonywanych w tym samym czasie, i na poziomie procesora, gdy różne wątki tej samej aplikacji są wykonywane przez różne procesory. Hierarchiczna pamięć jest odpowiedzią na podstawowe ograniczenie polegające na tym, że potrafimy konstruować bardzo szybkie lub bardzo wielkie pamięci, ale nie takie, które byłyby jednocześnie szybkie i wielkie.

Równoległość

Wszystkie nowoczesne mikroprocesory wykorzystują równoległość na poziomie instrukcji. Jednak ten typ równoległości jest zwykle niewidoczny dla programisty. Programy pisane są tak, jakby wszystkie instrukcje były wykonywane sekwencyjnie; sprzęt dynamicznie sprawdza zależności między instrukcjami w sekwencyjnym strumieniu i wywołuje je równolegle, gdy to możliwe. W niektórych przypadkach komputer zawiera sprzętowy program szeregujący, który może zmienić uporządkowanie instrukcji w celu zwiększenia równoległości wykonywania programu. Niezależnie od tego, czy sprzęt zmienia kolejność instrukcji, czy nie, kompilatory mogą ją zmieniać w celu bardziej efektywnego wykorzystania równoległości na poziomie instrukcji.

Równoległość na poziomie instrukcji może również jawnie występować w zbiorze instrukcji. Komputery VLIW (Very Long Instruction Word – bardzo długie słowo instrukcji) zawierają instrukcje, które mogą wywoływać wiele operacji równoległych. Dobrze znanym przykładem takiej architektury jest Intel IA64. Wszystkie wysoko wydajne mikroprocesory ogólnego przeznaczenia zawierają również instrukcje, które pozwalają operować na wektorach danych jednocześnie. Opracowane zostały techniki kompilacji automatycznie generujące kod dla takich maszyn na podstawie programów sekwencyjnych.

Systemy wieloprocesorowe stały się ponadto powszechne. Nawet komputery osobiste zwykle zawierają wiele procesorów (rdzeni). Programiści mogą sami pisać wielowątkowy kod dla systemów wieloprocesorowych, ale kod równoległy może być też automatycznie generowany przez kompilator z konwencjonalnych programów sekwencyjnych. Tego typu kompilator ukrywa przed programistą szczegóły wyszukiwania równoległości w programie, rozpraszania obliczeń w całej maszynie i minimalizowanie niezbędnej synchronizacji i komunikacji między procesorami. Wiele aplikacji inżynierskich i naukowych jest bardzo obciążających obliczeniowo i mogą ogromnie skorzystać na przetwarzaniu równoległym. Zostały więc opracowane techniki zrównoleglania pozwalające automatycznie tłumaczyć sekwencyjne programy naukowe na kod wieloprocesorowy.

Hierarchiczne pamięci

Hierarchia pamięci składa się z wielu poziomów pamięci o różnej prędkości i rozmiarach, przy czym poziomy najbliższe procesorowi są najszybsze, ale najmniejsze. Średni czas dostępu do pamięci w programie zmniejsza się, jeśli większość tych dostępów jest realizowana przez szybsze poziomy tej hierarchii. Zarówno równoległość, jak i dostępność hierarchii pamięci zwiększają potencjalną wydajność komputera, ale muszą zostać właściwie zaprzęgnięte przez kompilator, aby zapewnić rzeczywisty wzrost wydajności aplikacji.

Hierarchie pamięci występują we wszystkich typach komputerów. Procesor zazwyczaj ma niewielką liczbę rejestrów mieszczących po kilka bajtów, kilka poziomów pamięci podręcznej (buforów) zawierających od kilo- do megabajtów, pamięć fizyczną o wielkości od mega- do gigabajtów, a na koniec zasadniczy magazyn (pamięć dyskową) o rozmiarach gigabajtów i powyżej. Odpowiednio prędkość dostępu między sąsiadującymi poziomami hierarchii może różnić się od dwóch do trzech rzędów wielkości. Wydajność systemu jest często ograniczana nie tyle przez szybkość procesora, ile przez sprawność podsystemu pamięci. Podczas gdy tradycyjne kompilatory skupiały się na optymalizowaniu wykonywaniu kodu przez procesor, obecnie większy nacisk jest kładziony na zapewnienie większej efektywności hierarchicznej pamięci.

Wydajne używanie rejestrów jest zapewne najważniejszym pojedynczym problemem przy optymalizacji programu. W odróżnieniu od rejestrów, którymi programy mogą jawnie zarządzać, bufory i pamięci fizyczne są ukryte przed zestawami instrukcji i są zarządzane przez sprzęt. Można się przekonać, że zasady zarządzania buforowaniem implementowane przez sprzęt nie są efektywne w niektórych przypadkach, szczególnie w kodzie naukowym, który wykorzystuje wielkie struktury danych (zazwyczaj tablice). Możliwe jest usprawnienie efektywności hierarchii pamięci przez zmianę układu danych lub zmianę kolejności instrukcji odwołujących się do tych danych. Możemy również zmienić układ samego kodu, aby poprawić wydajność buforów instrukcji.

1.5.3. Projekty nowych architektur komputerów

We wczesnych latach projektowania komputerów kompilatory były zazwyczaj tworzone dopiero po zbudowaniu samych maszyn. To się zmieniło. Ponieważ normą jest dziś programowanie w językach wysokiego poziomu, wydajność systemu komputerowego jest determinowana nie tylko przez jego fizyczną prędkość, lecz także tym, jak dobrze kompilatory mogą wykorzystać jego funkcjonalności. Tym samym przy projektowaniu nowoczesnych architektur komputerów kompilatory są tworzone na etapie projektowania procesora, skompilowany zaś kod uruchamiany na symulatorach służy do oceniania proponowanych funkcji architektonicznych.

RISC

Jednym z najbardziej znanych przykładów tego, jak kompilatory wpływały na projektowanie komputerów, było wynalezienie architektury RISC (Reduced Instruction-Set Computer – komputer o zredukowanej liście rozkazów). Przed dokonaniem tego wynalazku panował trend projektowania coraz bardziej złożonych zestawów rozkazów mających na celu ułatwienie programowania w asemblerze; architektury te znane są pod nazwą CISC (Complex Instruction-Set Computer – komputer o złożonej liście rozkazów). Na przykład zbiór rozkazów CISC może zawierać złożone tryby adresowania pamięci w celu obsłużenia dostępu do struktur danych i wywoływania procedur oszczędzających wykorzystanie rejestrów i przekazywanie parametrów przez stos.

Optymalizacje kompilatora często mogą zredukować te rozkazy do niewielkiej liczby prostszych operacji przez wyeliminowanie nadmiarowości między złożonymi rozkazami. Stąd pożądane jest budowanie prostych zestawów rozkazów; kompilatory mogą używać ich wydajnie i optymalizacja dla takiego sprzętu jest znacznie łatwiejsza.

Większość znanych architektur procesorów ogólnego stosowania, w tym PowerPC, SPARC, MIPS, Alpha i PA-RISC, opartych jest na koncepcji RISC. Wprawdzie architektura x86 – najpopularniejszego mikroprocesora – ma zestaw instrukcji CISC, jednak wiele koncepcji wypracowanych dla maszyn RISC zostało użytych w implementacji samego procesora. Co więcej, najbardziej efektywną metodą użycia wysoko wydajnego komputera w architekturze x86 jest używanie jedynie prostych rozkazów procesora.

Architektury specjalizowane

W minionych czterdziestu latach zaproponowano wiele różnych koncepcji architektonicznych. Możemy do nich zaliczyć maszyny do przepływu danych, procesory wektorowe, maszyny VLIW (Very Long Instruction Word), macierze procesorów SIMD (Single Instruction, Multiple Data), tablice systoliczne, układy wieloprocesorowe z pamięcią wspólną lub rozproszoną. Wynajdywaniu każdej z tych koncepcji architektonicznych towarzyszyły badania i projektowanie odpowiadającej im technologii kompilatorów.

Niektóre z tych koncepcji znalazły zastosowanie w projektowaniu systemów wbudowanych. Ponieważ cały system komputerowy może zmieścić się w pojedynczym chipie, procesory nie muszą być już uniwersalnymi jednostkami, ale mogą zostać przycięte w celu uzyskania najlepszego stosunku kosztu do wydajności dla konkretnego zastosowania. Zatem, w odróżnieniu od procesorów ogólnego zastosowania, gdzie dążenie do uzyskania korzyści skali doprowadziło do upodobniania się architektur komputerów, procesory specjalizowane prezentują wielką różnorodność architektur. Technologie kompilatorów są potrzebne nie tylko w celu zapewnienia obsługi programowania dla tych architektur, lecz także do oceniania proponowanych projektów.

1.5.4. Tłumaczenie programów

Choć zazwyczaj pod pojęciem kompilowania rozumiemy tłumaczenie z języka wysokiego poziomu do poziomu kodu maszynowego, ta sama technologia może zostać zastosowana do tłumaczenia między różnymi językami. Poniżej przedstawiamy kilka ważnych zastosowań technik tłumaczenia programów.

Tłumaczenia binarne

Technologia kompilatorów może zostać użyta do przetłumaczenia kodu binarnego przeznaczonego dla jednej maszyny na kod innej, pozwalając komputerowi na uruchamianie programów oryginalnie skompilowanych dla innego zestawu instrukcji. Tłumaczenia binarne są używane przez różne firmy komputerowe w celu powiększenia dostępności oprogramowania dla ich maszyn.

W szczególności, ze względu na dominację architektury x86 na rynku komputerów osobistych, większość tytułów oprogramowania dostępnych jest w postaci kodu x86. Translatory binarne zostały opracowane w celu przekonwertowania kodu x86 na kod procesorów Alpha i Sparc. Tłumaczenia binarne zostały również użyte przez firmę Transmeta Inc. w jej implementacji zestawu instrukcji x86. Zamiast wykonywania zestawu złożonych instrukcji x86 bezpośrednio przez sprzęt, procesor Transmeta Crusoe jest procesorem o architekturze VLIW, którego działanie opiera się na binarnym tłumaczeniu kodu x86 na natywny kod VLIW.

Tłumaczenia binarne mogą być również wykorzystane w celu zapewnienia wstecznej kompatybilności. Gdy w roku 1994 w komputerach Apple Macintosh zmieniono procesory Motorola MC 68040 na PowerPC, użyto tłumaczenia binarnego w celu umożliwienia wykonywania starszego kodu MC 68040 przez procesory PowerPC.

Synteza sprzętowa

Nie tylko większość oprogramowania pisana jest w językach wysokiego poziomu; nawet projekty sprzętu są najczęściej opisywane w językach opisu sprzętu na wysokim poziomie abstrakcji, takich jak Verilog i VHDL (Very high-speed integrated circuit Hardware Description Language). Projekty sprzętu są typowo opisywane na poziomie przesłań międzyrejestrowych (*register transfer level* – RTL), gdzie zmienne reprezentują rejestry, a wyrażenia logikę kombinatoryczną. Narzędzia syntezy sprzętowej automatycznie tłumaczą opisy RTL na bramki, które są następnie mapowane na tranzystory, a w ostateczności na fizyczny układ. W odróżnieniu od kompilatorów języków programowania, narzędzia te często potrzebują godzin na optymalizację układu. Istnieją też techniki tłumaczenia projektów na wyższych poziomach, takich jak poziom behawioralny lub funkcjonalny.

Interpretery zapytań bazodanowych

Poza specyfikowaniem oprogramowania i sprzętu, języki są użyteczne w wielu innych zastosowaniach. Dobrym przykładem są języki zapytań, a zwłaszcza SQL (Structured Query Language), służące do przeszukiwania baz danych. Zapytania bazodanowe składają się z predykatów zawierających operatory relacyjne i logiczne. Mogą one być interpretowane lub kompilowane w polecenia przeszukujące bazy danych pod kątem rekordów spełniających te predykaty.

Skompilowane symulacje

Symulacja to ogólna technika wykorzystywana w wielu dyscyplinach naukowych i inżynierskich w celu lepszego zrozumienia jakiegoś zjawiska lub weryfikacji projektu. Wejście do symulatora zazwyczaj zawiera opis projektu i określone parametry wejściowe, dla których jest wykonywana dana symulacja. Symulacje mogą być bardzo kosztowne obliczeniowo. Zazwyczaj potrzebujemy zasymulować wiele możliwych alternatywnych projektów dla wielu różnych zestawów danych wejściowych i każdy taki eksperyment może wymagać wielu dni na wysoko wydajnym komputerze. Zamiast pisania symulatora interpretującego projekt szybsze jest skompilowanie tego projektu w celu utworzenia kodu maszynowego, który natywnie będzie symulował ten określony projekt. Skompilowane symulacje mogą działać o rzędy wielkości szybciej niż w przypadku podejścia opartego na interpreterze. Skompilowane symulacje są wykorzystywane w wielu zaawansowanych narzędziach symulujących projekty napisane w językach Verilog lub VHDL.

1.5.5. Narzędzia niezawodności oprogramowania

Programy są bez wątpienia najbardziej skompilowanymi artefaktami, jakie kiedykolwiek tworzono; składają się z wielu, wielu szczegółów, z których każdy musi być poprawny, aby program mógł działać bezbłędnie. W rezultacie błędy są wszechobecne w programach; błędy mogą położyć system, powodować zwracanie niewłaściwych wyników, sprawiać, że system jest podatny na ataki, a nawet doprowadzić do katastrofalnych awarii w systemach krytycznych. Podstawową techniką lokalizowania błędów w programach jest testowanie.

Interesującym i obiecującym podejściem uzupełniającym jest użycie analizy przepływu danych w celu statycznej lokalizacji błędów (czyli jeszcze przed uruchomieniem programu). Analiza przepływu danych pozwala znaleźć błędy na wszystkich możliwych ścieżkach wykonania, a nie tylko na tych, które następują dla określonych zestawów danych wejściowych, jak w przypadku testowania programu. Wiele z technik analizy przepływu danych, oryginalnie stworzonych z myślą o optymalizacji w kompilatorach, może zostać użytych do tworzenia narzędzi wspomagających programistów w zadaniach inżynierii oprogramowania.

Problem odszukania wszystkich błędów w programie jest nierozstrzygalny. Analiza przepływu danych może ostrzegać programistów o wszystkich możliwych instrukcjach mogących generować określone kategorie błędów. Jednak jeśli większość z tych ostrzeżeń będzie fałszywymi alarmami, użytkownicy nie będą używać takiego narzędzia. Stąd praktyczne detektory błędów najczęściej nie są ani dokładne, ani kompletne. Innymi słowy, mogą one nie znaleźć wszystkich błędów w programie i nie ma gwarancji, że raportowane błędy są rzeczywistymi błędami. Niemniej jednak rozwinięto rozmaite analizy statyczne i pokazały one swoją skuteczność w wyszukiwaniu błędów w rzeczywistych programach, takich jak dereferencje do pustych lub zwolnionych wskaźników. Fakt, że detektory błędów mogą być niedokładne, sprawia, że zasadniczo różnią się od optymalizacji. Optymalizator musi zachowywać się konserwatywnie i nie może zmienić semantyki programu w żadnych okolicznościach.

W celu dopełnienia tego podrozdziału należy wspomnieć wiele sposobów, którymi analiza programów, oparta na technikach wynalezionych oryginalnie w celu optymalizowania kodu w kompilatorach, poprawiła wydajność oprogramowania. Szczególnie ważne są techniki, które potrafią statycznie wykryć potencjalną podatność zabezpieczeń w programie.

Sprawdzanie typów

Sprawdzanie zgodności typów jest efektywną i dobrze ugruntowaną techniką wyłapywania niespójności w programach. Może zostać użyte do wychwytywania błędów, na przykład wtedy, gdy operacja zostanie zastosowana wobec obiektu niewłaściwego typu lub gdy parametry przekazane do procedury nie pasują do sygnatury tej procedury. Analiza programu może pójść jeszcze dalej w wykrywaniu błędów typowania przez analizę przepływu danych przez program. Na przykład jeśli do wskaźnika przypiszemy null, po czym natychmiast się do niego odwołamy, to bez wątpienia mamy do czynienia z błędem.

Tej samej technologii można użyć do wykrywania rozmaitych luk zabezpieczeń, dzięki którym atakujący dostarcza łańcuch znaków lub inne dane, które są nieostrożnie wykorzystywane przez program. Łańcuch dostarczany przez użytkownika może zostać opatrzony etykietą typu „niebezpieczny". Jeśli ten łańcuch nie zostanie sprawdzony pod kątem właściwego formatu, pozostaje „niebezpieczny", a jeśli taki tekst jest w stanie wpływać na sterowanie przepływem kodu w jakimś punkcie programu, wówczas jest to potencjalna słabość zabezpieczeń.

Sprawdzanie granic

Łatwiejsze jest popełnianie błędów, gdy programuje się w języku niższego poziomu, niż w językach wyższych poziomów. Na przykład wiele włamań i luk zabezpieczeń w systemach powodowanych jest przez przepełnienia bufora w programach napisanych w C. Ponieważ język C nie zawiera wbudowanego

mechanizmu sprawdzania granic tablic, to na programistę spada zapewnienie, że odwoływanie się do tablicy nie będzie sięgało poza jej granicę. Jeśli nie wykonamy sprawdzenia, czy dane dostarczone przez użytkownika mogą przepełnić bufor, program może zostać oszukany tak, by przechował dane użytkownika poza buforem. Atakujący może zmanipulować dane wejściowe w taki sposób, aby spowodować niewłaściwe działanie programu i przełamać zabezpieczenia systemu. Opracowano wiele technik wyszukiwania przepełnień bufora w programach, ale z ograniczonymi sukcesami.

W przypadku gdy program został napisany w bezpiecznym języku, który zawiera automatyczne sprawdzanie granic, problem taki nie występuje. Ta sama analiza przepływu danych, która jest używana do eliminowania nadmiarowych sprawdzań zakresów, może być również użyta do lokalizowania przepełnień bufora. Główna różnica polega jednak na tym, że pominięcie wyeliminowania sprawdzania zakresów spowoduje jedynie niewielki dodatkowy koszt w czasie wykonywania programu, podczas gdy niewykrycie potencjalnego przepełnienia bufora może skompromitować zabezpieczenia systemu.

Choć wystarczające jest użycie prostych technik do optymalizowania sprawdzania zakresów, zaawansowane analizy, takie jak śledzenie wartości wskaźników między procedurami, są konieczne, jeśli chcemy uzyskać wysokiej jakości wyniki w narzędziach wykrywania błędów.

Narzędzia zarządzania pamięcią

Odśmiecanie pamięci (*garbage collection* – GC) to inny doskonały przykład nieuniknionego kompromisu między wydajnością a połączeniem łatwości programowania i niezawodności programu. Automatyczne zarządzanie pamięcią zamazuje wszelkie błędy zarządzania pamięcią (czyli „wycieki pamięci"), które są głównym źródłem problemów w programach pisanych w C i C++. Opracowano rozmaite narzędzia pomagające w wyszukiwaniu błędów zarządzania pamięcią. Na przykład Purify jest szeroko używanym narzędziem dynamicznie wychwytującym błędy zarządzania pamięcią, gdy tylko wystąpią. Istnieją też narzędzia, które pomagają statycznie identyfikować niektóre z takich problemów.

1.6. Podstawy języków programowania

W tej części zajmiemy się najważniejszą terminologią i rozróżnieniami, które występują w studiach nad językami programowania. Nie jest naszym celem omawianie wszystkich koncepcji ani wszystkich popularnych języków programowania. Zakładamy, że Czytelnik zna przynajmniej jeden język spośród „wielkiej czwórki" – C, C++, C# lub Java – a być może zetknął się również z innymi językami.

1.6.1. Rozróżnienie statyczny/dynamiczny

Wśród najważniejszych problemów, z którymi musimy się zmierzyć przy projektowaniu kompilatora dla określonego języka, jest ten, jakie decyzje może podejmować kompilator na temat tworzonego programu. Z jednej strony jeśli język używa zasady pozwalającej kompilatorowi na rozstrzygnięcie jakiegoś problemu, wówczas mówimy, że język ten używa *zasady statycznej* lub że dylemat zostanie rozstrzygnięty *w czasie kompilacji*. Z drugiej strony zasada pozwalająca na podejmowanie decyzji tylko podczas działania programu nazywana jest *zasadą dynamiczną* lub wymagającą decyzji *w czasie wykonywania*.

Problemem, nad którym powinniśmy się skoncentrować, jest zasięg deklaracji. *Zasięgiem* deklaracji x nazywamy obszar programu, w którym użycie x odwołuje się do tej deklaracji. Język używa *zasięgu statycznego* lub *leksykalnego*, jeśli możliwe jest określenie zasięgu deklaracji przez samo przejrzenie kodu programu. W przeciwnym razie język używa *zasięgu dynamicznego*. W tym przypadku podczas działania programu to samo użycie x może odnosić się do dowolnej z wielu różnych deklaracji x.

Większość języków, takich jak C i Java, używa zasięgu statycznego. Zagadnienie to omówimy bliżej w podrozdziale 1.6.3.

Przykład 1.3: Jako inny przykład rozróżnienia między statycznym a dynamicznym rozważmy użycie terminu „static", tak jak jest stosowany do danych w deklaracji klasy w Javie. W języku Java zmienna jest nazwą lokalizacji w pamięci, służącej do przechowania wartości danych. W tym przypadku „static" nie odnosi się do zasięgu zmiennej, ale do możliwości zdeterminowania przez kompilator lokalizacji w pamięci, w której można znaleźć zadeklarowaną zmienną. Deklaracja taka jak

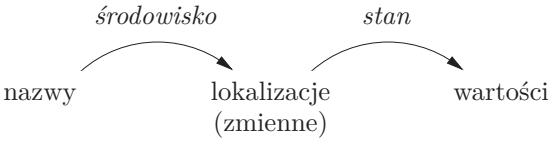
```
public static int x;
```

sprawia, że x jest *zmienną klasową* i że istnieje tylko jedna kopia x, niezależnie od tego, jak wiele obiektów tej klasy zostanie utworzonych. Co więcej, kompilator może ustalić lokalizację w pamięci, w której to całkowite x będzie przechowywane. Dla kontrastu, gdybyśmy pominęli „static" w tej deklaracji, każdy obiekt klasy miałby swoją własną lokalizację, w której przechowywane będzie x, i kompilator nie może określić wszystkich tych miejsc zawczasu przed uruchomieniem programu. ■

1.6.2. Środowiska i stany

Inne ważne rozróżnienie, które musimy poczynić przy omawianiu języków programowania, jest to, czy zmiany występujące w czasie działania programu wpływają na wartości elementów danych, czy też na interpretację nazw dla tych danych. Na przykład wykonanie wyrażenia przypisania, takiego jak `x = y + 1`, zmienia wartość wskazywaną przez nazwę x. Mówiąc precyzyjniej, przypisanie zmienia wartość w jakiejkolwiek lokalizacji, która jest oznaczona przez x.

Mniej oczywiste może być to, że lokalizacja oznaczona przez x może ulec zmianie w czasie działania programu. Przykładowo, jak pokazaliśmy w przykładzie 1.3, jeśli x nie jest zmienną statyczną (inaczej „klasową"), wówczas każdy obiekt tej klasy ma swoją własną lokalizację dla wystąpienia zmiennej x. W takim przypadku przypisanie do x może zmienić pewną z tych „wystąpieniowych" zmiennych, zależnie od obiektu, do którego zastosowana będzie metoda zawierająca to przypisanie.



RYSUNEK 1.8: Dwufazowe mapowanie od nazw do wartości

Powiązanie nazw z lokalizacjami w pamięci (magazynie) i następnie z ich wartościami można opisać jako dwa odwzorowania, które mogą zmieniać się w trakcie działania programu (patrz rysunek 1.8):

1. *Środowisko* (*environment*) to odwzorowanie nazw na lokalizacje w pamięci. Ponieważ zmienne odnoszą się do lokalizacji („l-wartości" w terminologii języka C), możemy alternatywnie zdefiniować środowisko jako odwzorowanie nazw na zmienne.
2. *Stan* (*state*) to odwzorowanie lokalizacji w pamięci na ich wartości. Inaczej mówiąc, stan odwzorowuje l-wartości na odpowiadające im r-wartości w terminologii C.

Środowiska zmieniają się zgodnie z regułami zasięgu języka.

Przykład 1.4: Rozważmy fragment programu C pokazany na rysunku 1.9. Zmienna całkowita i jest zadeklarowana jako zmienna globalna, jak również jako zmienna lokalna funkcji f. Gdy wykonywana jest funkcja f, środowisko dopasowuje to, do czego odwołuje się nazwa i, tak by wskazywała lokalizację zarezerwowaną dla i jako lokalnej dla f, i dowolne użycie i, jak pokazane jawnie przypisanie i = 3, odwołuje się do tej lokalizacji. Typowo lokalna zmienna i otrzymuje miejsce w stosie czasu wykonania.

Ilekroć funkcja g (inna niż f) jest wykonywana, użycie zmiennej i nie może odwołać się do tego i, które jest lokalne dla f. Użycie nazwy i w g musi być w zasięgu jakiejś innej deklaracji i. Przykład pokazuje wyrażenie x = i+1, które znajduje się wewnątrz jakiejś procedury, która definicja nie jest pokazana. Zmienna i w wyrażeniu $i + 1$ domyślnie odnosi się do globalnej zmiennej i. Jak w większości języków, deklaracje zmiennych w C muszą poprzedzać ich użycie, zatem funkcja umieszczona przed globalną deklaracją i nie może się odwoływać do tej zmiennej. ■

Odwzorowania środowiska i stanu pokazane na rysunku 1.8 są dynamiczne, ale istnieje kilka wyjątków:

```
...
int i;                  /* globalne i          */
...
void f(···) {
      int i;            /* lokalne i           */
      ...
      i = 3;            /* użycie lokalnego i  */
      ...
}
...
      x = i + 1;        /* użycie globalnego i */
```

RYSUNEK 1.9: Dwie deklaracje nazwy i

1. Statyczne albo dynamiczne wiązanie nazw z lokalizacjami. Większość wiązań nazw z lokalizacjami jest dynamicznych i będziemy omawiać kilka podejść do tego zagadnienia w dalszej części podrozdziału. Jednak niektóre deklaracje, takie jak globalne i na rysunku 1.9, mogą otrzymać lokalizację w pamięci tylko raz na zawsze, gdy kompilator będzie generował kod wynikowy[2].
2. Statyczne albo dynamiczne wiązanie lokalizacji z wartościami. Wiązanie lokalizacji z wartościami (druga faza na rysunku 1.8) również jest w ogólności dynamiczne, ponieważ nie możemy przewidzieć wartości, która będzie umieszczona w tej lokalizacji, dopóki nie uruchomimy programu. Wyjątkiem są zadeklarowane stałe. Na przykład definicja w języku C

   ```
   #define ARRAYSIZE 1000
   ```

   statycznie wiąże nazwę `ARRAYSIZE` z wartością 1000. Możemy ustalić to powiązanie, przyglądając się wyrażeniu, i wiemy, że nie jest możliwe, aby wiązanie to zmieniło się podczas wykonywania programu.

1.6.3. Statyczny zasięg i struktura blokowa

Większość języków, w tym C i jego rodzina, używa zasięgu statycznego. Reguły zasięgu w języku C bazują na strukturze programu; zasięg deklaracji jest determinowany niejawnie przez miejsce, w którym deklaracja pojawia się w programie. Późniejsze języki, takie jak C++, Java i C#, dodatkowo udostępniają

[2] Z technicznego punktu widzenia kompilator C przypisze lokalizację w pamięci wirtualnej dla globalnej zmiennej i, pozostawiając przypisanie określonego miejsca w pamięci fizycznej maszyny programowi ładującemu i systemowi operacyjnemu. Jednak nie musimy martwić się o problemy „relokacji", gdyż takowe nie mają żadnego wpływu na kompilację. Zamiast tego traktujemy przestrzeń adresową, której kompilator używa dla swojego kodu wynikowego, tak jakby były to lokalizacje w pamięci fizycznej.

Nazwy, identyfikatory i zmienne

Choć terminy „nazwa" i „zmienna" często odnoszą się do tej samej rzeczy, należy używać ich z rozwagą, aby zachować rozróżnienie między nazwami czasu kompilacji a lokalizacjami czasu wykonania oznaczanymi przez te nazwy.

Identyfikator to łańcuch znaków, zazwyczaj liter lub cyfr, który odnosi się do (identyfikuje) pewnego bytu, taki jak obiekt danych, procedura, klasa lub typ. Wszystkie identyfikatory są nazwami, ale nie każda nazwa jest identyfikatorem. Nazwy mogą być również wyrażeniami. Na przykład nazwa $x.y$ może oznaczać pole y struktury wskazywanej przez x. W tym przypadku x oraz y są identyfikatorami, podczas gdy $x.y$ jest nazwą, ale nie identyfikatorem. Złożone nazwy, takie jak $x.y$, określa się mianem *nazw kwalifikowanych*.

Zmienna odwołuje się do określonej lokalizacji w pamięci. Często spotykane jest deklarowanie tego samego identyfikatora więcej niż jeden raz; każda taka deklaracja wprowadza nową zmienną. Nawet jeśli identyfikator został zadeklarowany tylko raz, to na przykład identyfikator lokalny w procedurze rekurencyjnej będzie się odnosił do różnych lokalizacji w pamięci w różnych momentach.

jawną kontrolę nad zasięgami przez użycie słów kluczowych, takich jak **public, private** i **protected**.

W tym punkcie rozważymy reguły zasięgu statycznego dla języka blokowego, przy czym *blokiem* nazywamy zgrupowanie deklaracji i instrukcji. Język C do wyznaczania granic bloku używa nawiasów klamrowych **{** i **}**; alternatywne stosowanie słów kluczowych **begin** oraz **end** do tego samego celu sięga wstecz czasów Algolu.

Przykład 1.5: W pierwszym przybliżeniu zasada zasięgu statycznego w języku C wygląda następująco:

1. Program C składa się z sekwencji deklaracji zmiennych i funkcji najwyższego poziomu.
2. Funkcje mogą zawierać w sobie deklaracje zmiennych, przy czym zmienne te mogą obejmować zmienne lokalne i parametry. Zasięg każdej z takich deklaracji jest ograniczony do funkcji, w której występuje.
3. Zasięg deklaracji najwyższego poziomu nazwy x obejmuje cały dalszy ciąg programu po tej deklaracji z wyjątkiem tych instrukcji, które znajdują się wewnątrz funkcji również zawierających deklarację nazwy x.

Dodatkowe szczegóły dotyczące zasady zasięgu statycznego w języku C dotyczą deklaracji zmiennych wewnątrz wyrażeń. Przeanalizujemy takie deklaracje w przykładzie 1.6. ■

Procedury, funkcje i metody

Chcąc uniknąć powtarzania frazy „procedury, funkcje lub metody" za każdym razem, gdy będziemy chcieli mówić o podprogramie, który może zostać wywołany, zazwyczaj będziemy odnosić się do nich wszystkich jako „procedur". Wyjątkiem jest sytuacja, gdy będziemy omawiać programy w takich językach, jak C, które mają tylko funkcje – wówczas będziemy się odwoływać się do nich jako „funkcji". Analogicznie, przy omawianiu takiego języka, jak Java, w którym istnieją tylko metody, będziemy używać tego właśnie terminu.

Funkcja w ogólności zwraca wartość jakiegoś typu („typ zwracany"), podczas gdy procedura może nie zwracać żadnej wartości. C i podobne języki, które zawierają tylko funkcje, traktują procedury jako funkcje o specjalnym typie zwracanym „void", aby zaznaczyć brak zwracanej wartości. W językach zorientowanych obiektowo, jak Java i C++, używany jest termin „metody". Mogą one zachowywać się jak funkcje lub procedury, ale są powiązane z określoną klasą.

W języku C składnia definiująca bloki jest określana następująco:

1. Jednym z typów instrukcji jest blok. Bloki mogą pojawiać wszędzie tam, gdzie mogą występować inne typy instrukcji, takie jak instrukcje przypisania.
2. Blok jest sekwencją deklaracji, po których następuje sekwencja instrukcji, całość jest otoczona nawiasami klamrowymi.

Zauważmy, że ta składnia pozwala na zagnieżdżanie bloków wewnątrz siebie. To zagnieżdżanie określane jest mianem *struktury blokowej*. Rodzina języków C ma strukturę blokową z tym wyjątkiem, że funkcji nie można definiować wewnątrz innej funkcji.

Powiemy, że deklaracja D „należy do" bloku B, jeśli B jest najbliższym zagnieżdżonym blokiem zawierającym D; inaczej mówiąc, D jest zlokalizowane wewnątrz B, ale nie w żadnym innym bloku zagnieżdżonym wewnątrz B.

Reguła zasięgu statycznego dla deklaracji zmiennych w językach o strukturze blokowej jest następująca: jeśli deklaracja D nazwy x należy do bloku B, wówczas zasięgiem D jest cały blok B z wyjątkiem dowolnych bloków B' zagnieżdżonych (do dowolnej głębokości) wewnątrz B, w których x zostało ponownie zadeklarowane. Ponowna deklaracja x w B' następuje wtedy, gdy istnieje jakaś deklaracja D' tej samej nazwy x należąca do B'.

Równoważny sposób wyrażenia tej reguły polega na skupieniu się na użyciu nazwy x. Niech $B_1, B_2, \ldots, B_k$ będą blokami otaczającymi określone użycie nazwy x, przy czym B_k jest blokiem najmniejszym, zagnieżdżonym wewnątrz B_{k-1}, który jest zagnieżdżony wewnątrz bloku B_{k-2} i tak dalej. Szukamy największego i takiego, że istnieje deklaracja nazwy x należąca do B_i. Nasze

użycie nazwy x odnosi się do deklaracji zawartej w B_i. Alternatywnie można powiedzieć, że to użycie nazwy x należy do zasięgu deklaracji zawartej w B_i.

Przykład 1.6: Program w języku C++ pokazany na rysunku 1.10 zawiera cztery bloki z wieloma definicjami zmiennych a i b. Dla łatwiejszej orientacji każda deklaracja inicjuje swoją zmienną wartością liczbową odpowiadającą blokowi, do którego należy.

```
main() {
    int a = 1;                                   B1
    int b = 1;
    {
        int b = 2;                               B2
        {
            int a = 3;                           B3
            cout « a « b;
        }
        {
            int b = 4;                           B4
            cout « a « b;
        }
        cout « a « b;
    }
    cout « a « b;
}
```

RYSUNEK 1.10: Bloki w programie w języku C++

Na przykład rozważmy deklarację int `int a = 1` w bloku B_1. Jej zasięgiem jest cały blok B_1 z wyjątkiem tych zagnieżdżonych (być może dość głęboko) bloków wewnątrz B_1, które mają własne deklaracje zmiennej a. B_2, zagnieżdżony bezpośrednio w B_1, nie zawiera takiej deklaracji, ale blok B_3 już tak. B_4 nie zawiera deklaracji zmiennej a, zatem blok B_3 jest jedynym miejscem w całym programie, które jest poza zasięgiem deklaracji zmiennej zawartej w bloku B_1. Inaczej mówiąc, zasięg ten zawiera blok B_4 i cały blok B_2 z wyjątkiem tej części B_2, która jest wewnątrz B_3. Zasięgi wszystkich pięciu deklaracji zostały podsumowane na rysunku 1.11.

Deklaracja	Zasięg
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

RYSUNEK 1.11: Zasięgi deklaracji z przykładu 1.6

Spoglądając na to zagadnienie z innej strony, możemy rozważyć instrukcję wyjścia zawartą w bloku B_4 i powiązać użyte tu zmienne a i b z odpowiednimi deklaracjami. Lista otaczających bloków w kolejności rosnących rozmiarów to $B_4; B_2; B_1$. Zwróćmy uwagę, że B_3 nie otacza analizowanego punktu programu. Blok B_4 zawiera deklarację b, zatem to do tej deklaracji odnosi się to użycie zmiennej b i wypisana wartość zmiennej b to 4. Jednak w B_4 nie ma deklaracji zmiennej a, zatem kolejnym miejscem do sprawdzenia jest B_2. Ten blok również nie zawiera deklaracji a, zatem przechodzimy do B_1. Szczęśliwie tu znajdujemy deklarację `int a = 1`, zatem wypisana wartość a to 1. Gdyby nie istniała taka deklaracja, program byłby błędny. ■

1.6.4. Jawna kontrola dostępu

Klasy i struktury wprowadzają nowe pojęcie zasięgu dla swoich elementów składowych. Jeśli p jest obiektem klasy zawierającej pole (członka) x, wówczas użycie x w wyrażeniu $p : x$ odnosi się do pola x w definicji klasy. Analogicznie do struktury blokowej, zasięg deklaracji pola x w klasie C rozszerza się na dowolną klasę pochodną C', chyba że C' zawiera lokalną deklarację tej samej nazwy x.

Przez użycie takich słów kluczowych, jak **public, private** i **protected**, języki zorientowane obiektowo takie jak C++ lub Java zapewniają jawną kontrolę nad dostępem do nazw elementów składowych klasy bazowej. Te słowa kluczowe wspierają hermetyzację przez ograniczanie dostępu. Nazwy prywatne mają więc celowo narzucony zasięg, który obejmuje tylko deklaracje metod i definicje powiązane z tą samą klasą oraz klasami „zaprzyjaźnionymi" (termin używany w C++). Nazwy chronione są dostępne dla klas pochodnych. Nazwy publiczne są dostępne z dowolnego miejsca – z wnętrza klasy i spoza niej.

W C++ definicja klasy może zostać oddzielona od definicji części lub wszystkich jej metod. Tym samym dla nazwy x powiązanej z klasą C może istnieć region kodu leżący poza jej zasięgiem, po którym następuje inny region (definicja metody) znajdujący się w jej zasięgu. W istocie regiony wewnątrz zasięgu i poza nim mogą się przeplatać wzajemnie, dopóki wszystkie metody nie zostaną zdefiniowane.

1.6.5. Zasięg dynamiczny

Ujmując rzecz technicznie, dowolna zasada zasięgu jest dynamiczna, jeśli jest oparta na czynniku lub czynnikach, które mogą być znane jedynie podczas działania programu. Jednak termin *zasięg dynamiczny* zazwyczaj odnosi się do następującej zasady: użycie nazwy x odwołuje się do deklaracji x w ostatnio wywołanej i jeszcze nie zakończonej procedurze zawierającej taką deklarację. Dynamiczne definiowanie zasięgu tego rodzaju występuje tylko w szczególnych sytuacjach. Powinniśmy rozważyć dwa przykłady zasad dynamicznych: rozwijanie makr w preprocesorze C oraz rozpoznawanie metod w programowaniu obiektowym.

Deklaracje i definicje

Pozornie podobne terminy „deklaracja" i „definicja" używane w koncepcjach języków programowania w rzeczywistości są zupełnie różne. Deklaracje informują nas o typach pewnych bytów, podczas gdy definicje mówią nam o ich wartościach. Zatem int i jest deklaracją zmiennej i, podczas gdy i = 1 jest jej definicją.

Rozróżnienie to jest bardziej znaczące, gdy mamy do czynienia z metodami lub innymi procedurami. W C++ metoda jest deklarowana w definicji klasy przez podanie typów argumentów i wyniku tej metody (często jest to nazywane sygnaturą metody). Następnie, w innym miejscu metoda jest definiowana, czyli podawany jest kod wykonania tej metody. Analogicznie często spotykane jest definiowanie funkcji C w jednym pliku i zadeklarowanie jej w innych plikach, w których ta funkcja jest używana.

Przykład 1.7: W programie w języku C pokazanym na rysunku 1.12 identyfikator a jest makrem, które zawiera wyrażenie $(x + 1)$. Ale czym jest x? Nie jesteśmy w stanie ustalić wartości x statycznie, czyli analizując tekst programu.

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

RYSUNEK 1.12: Makro, którego nazwy muszą mieć zasięg dynamiczny

W istocie, aby móc zinterpretować x, musimy użyć zwyczajowej reguły zasięgu dynamicznego. Trzeba zbadać wszystkie wywołania funkcji, które są aktualnie aktywne, i wybrać ostatnio wywołaną z nich, która zawiera deklarację zmiennej x. To do tej deklaracji odnosi się użycie zmiennej x.

W przykładzie z rysunku 1.12 funkcja *main* najpierw wywołuje funkcję b. W trakcie wykonywania wypisuje ona wartość makra a. Ponieważ a musi być zastąpione zapisem $(x + 1)$, rozwiązujemy to użycie zmiennej x przez deklarację `int x = 1` w funkcji b. Powodem jest to, że funkcja b zawiera deklarację x, zatem wyrażenie $(x + 1)$ w `printf` w funkcji b odnosi się do tego x. Tym samym wartość wypisana to 2.

Po zakończeniu działania b i wywołaniu funkcji c ponownie potrzebujemy wypisać wartość makra a. Jednak tym razem jedynym dostępnym x jest globalnie zdefiniowane x. Wyrażenie `printf` w c odnosi się zatem do tej deklaracji x i zostanie wypisana wartość 3. ■

Analogia między zasięgami statycznymi i dynamicznymi

Choć może istnieć dowolna liczba zasad dynamicznych lub statycznych dla zasięgów, istnieje interesująca zależność między normalną (opartą na strukturze blokowej) regułą zasięgu statycznego i normalną zasadą dynamiczną. Zasadniczo, reguła dynamiczna dotyczy czasu, podczas gdy reguła statyczna – miejsca. Podczas gdy reguła statyczna nakazuje nam znaleźć deklarację, której jednostka (blok) najbliżej otacza fizyczną lokalizację użycia nazwy, reguła dynamiczna wymaga znalezienia deklaracji, której jednostka (wywołanie procedury) najbliżej otacza chwilę użycia.

Dynamiczne rozwiązywanie zasięgu jest również podstawową cechą procedur polimorficznych, czyli takich, które zawierają dwie lub więcej definicji dla tej samej nazwy, zależnych tylko od typów argumentów. W niektórych językach, takich jak ML (patrz podrozdział 7.3.3), możliwe jest statyczne zdeterminowanie typów dla wszystkich przypadków użycia nazw i w takim przypadku kompilator może zastąpić każde użycie nazwy procedury p referencją do kodu właściwej procedury. Jednak w innych językach, takich jak Java lub C++, istnieją sytuacje, w których kompilator nie może dokonać takiego rozstrzygnięcia.

Przykład 1.8: Cechą wyróżniającą programowania zorientowanego obiektowo jest możliwość wywoływania przez każdy obiekt odpowiedniej metody w odpowiedzi na komunikat. Innymi słowy, procedura wywoływana przy wykonywaniu $x : m()$ zależy od klasy obiektu wskazywanego przez x w tym momencie. Oto typowy przykład:

1. Mamy klasę C z metodą nazwaną $m()$.
2. D jest klasą pochodną C i zawiera swoją własną metodę o nazwie $m()$.
3. Następuje użycie metody m w postaci $x{:}m()$, gdzie x jest obiektem klasy C.

W normalnej sytuacji niemożliwe jest rozstrzygnięcie w czasie kompilacji, czy x będzie obiektem klasy C, czy też klasy pochodnej D. Jeśli zastosowanie metody występuje wielokrotnie, jest wysoce prawdopodobne, że część z nich będzie dotyczyło obiektów należących do klasy C, ale nie D, podczas gdy inne będą z klasy D. Jednak do czasu uruchomienia programu nie jest możliwe zdecydowanie, która definicja m jest tą właściwą w danym momencie. Tym samym kod wygenerowany przez kompilator musi ustalić klasę obiektu x i wywołać jedną lub drugą metodę m. ■

1.6.6. Mechanizmy przekazywania parametrów

Wszystkie języki programowania znają pojęcie procedury, ale mogą się różnić tym, jak te procedury uzyskują swoje argumenty. W tym podrozdziale zajmiemy się

tym, jak *argumenty* (*parametry aktualne*, czyli parametry użyte w wywołaniu procedury) są powiązane z *parametrami formalnymi* (tymi użytymi w definicji procedury). To, jaki mechanizm jest używany, determinuje, jak wywołująca sekwencja kodu traktuje parametry. Znakomita większość języków używa albo „przekazywania przez wartość", albo „przekazywania przez referencję", albo obu. Wyjaśnimy tutaj te terminy, a także jeszcze jedną metodę znaną jako „przekazywanie przez nazwę", która ma głównie znaczenie historyczne.

Przekazywanie przez wartość

W *przekazywaniu przez wartość* argument jest obliczany (jeśli jest to wyrażenie) lub kopiowany (jeśli jest to zmienna). Wartość ta jest umieszczana w lokalizacji należącej do odpowiadającego mu parametru formalnego wywoływanej procedury. Ta metoda jest używana w C i w Javie, a także jest często wybieraną opcją w C++, podobnie jak w większości innych języków. Przekazywanie przez wartość ma ten efekt, że wszystkie obliczenia wykorzystujące parametry formalne wykonywane przez wywoływaną procedurę są lokalne dla tej procedury i rzeczywiste parametry (w kodzie wywołującym) nie mogą być zmieniane.

Można jednak zauważyć, że w C możemy przekazać wskaźnik do zmiennej, aby umożliwić modyfikację tej zmiennej przez wywoływany podprogram. Analogicznie, przekazanie nazwy tablicy jako parametru w C, C++ lub Javie daje wywoływanej procedurze wskaźnik lub referencję do samej tablicy. Jeśli zatem a jest nazwą tablicy w wywołującej procedurze i zostanie przekazana przez wartość do odpowiadającego jej parametru formalnego x, wówczas przypisanie takie jak x[i] = 2 rzeczywiście zmieni element tablicy $a[i]$ na 2. Przyczyną takiego zachowania jest to, że choć x otrzymuje kopię wartości a, wartość ta jest w rzeczywistości wskaźnikiem do początku obszaru pamięci, w którym zlokalizowana jest tablica o nazwie a.

Analogicznie w Javie wiele zmiennych to w rzeczywistości referencje lub wskaźniki do rzeczy, na które pokazują. Obserwacja ta dotyczy tablic, łańcuchów oraz obiektów dowolnych klas. Mimo że Java używa wyłącznie przekazywania przez wartość, ilekroć przekazujemy nazwę obiektu do wywoływanej procedury, wartość otrzymana przez tę procedurę jest w istocie wskaźnikiem do tego obiektu. Tym samym wywoływana procedura jest w stanie wpływać na wartość samego obiektu.

Przekazywanie przez referencję

W *przekazywaniu przez referencję* adres rzeczywistego parametru jest przekazywany do wywoływanej procedury jako wartość odpowiadającego parametru formalnego. Użycie parametru formalnego w kodzie wywoływanym jest implementowany przez podążanie za tym wskaźnikiem do lokalizacji określonej przez kod wywołujący. Zmiany parametru formalnego wyglądają wówczas jak zmiany parametru aktualnego (argumentu).

Jeśli jednak argument jest wyrażeniem, wówczas oblicza się przed wywołaniem i jego wartość jest przechowywana w jego własnej lokalizacji. Zmiany parametru formalnego zmieniają wartość w tej lokalizacji, ale nie wpływają na dane w kodzie wywołującym.

Przekazywanie przez referencję jest używane dla parametrów „ref" w C++ i jest opcją dostępną w wielu innych językach. Jest niemal obowiązkowe, gdy parametr formalny jest wielkim obiektem, tablicą lub strukturą. Przyczyną jest to, że ścisłe przekazywanie przez wartość wymaga, aby kod wywołujący skopiował cały argument do przestrzeni należącej do odpowiadającego mu parametru formalnego. Kopiowanie to staje się kosztowną operacją, gdy argument jest duży. Jak zaznaczyliśmy przy omawianiu przekazywania przez wartość, języki takie jak Java rozwiązują problem przekazywania tablic, łańcuchów lub innych obiektów przez kopiowanie tylko referencji do tych obiektów. W efekcie Java zachowuje się tak, jakby używała przekazywania przez referencję dla dowolnych argumentów innych niż typy podstawowe, takie jak liczba całkowita lub rzeczywista.

Przekazywanie przez nazwę

Trzeci mechanizm – *przekazywanie przez nazwę* (call-by-name) – był używany we wczesnym języku programowania Algol 60. Wymaga on, aby kod wywoływany został wykonany, tak jakby aktualny parametr dosłownie zastąpił parametr formalny w kodzie wywoływanym, jakby było to makro zastępujące argument (wraz z przemianowaniem lokalnym nazw w wywoływanej procedurze, aby zachować ich rozróżnienie). Gdy argument jest wyrażeniem, a nie zmienną, występują pewne nieintuicyjne zachowania, co jest jednym z powodów, dla których mechanizm ten nie jest dziś preferowany.

1.6.7. Aliasowanie

Istnieje pewna interesująca konsekwencja przekazywania parametrów przez referencję lub jego symulowania, jak w Javie, gdzie referencje do obiektów są przekazywane przez wartość. Jest możliwe, że dwa parametry formalne mogą odwołać się do tej samej lokalizacji; takie zmienne nazywamy aliasami (synonimami). W rezultacie dowolne dwie zmienne, które, jak się wydaje, uzyskują swoje wartości z dwóch różnych parametrów formalnych, mogą również stać się swoimi wzajemnymi aliasami.

Przykład 1.9: Załóżmy, że a jest tablicą należącą do procedury p i procedura ta wywołuje inną procedurę $q(x, y)$ przez wywołanie $q(a, a)$. Załóżmy też, że parametry są przekazywane przez wartość, ale nazwy tablic są w istocie referencjami do lokalizacji, w których tablice są przechowywane, tak jak w C lub podobnych językach. Teraz x oraz y stają się wzajemnymi aliasami. Ważną kwestią jest to, że jeśli w procedurze q znajduje się przypisanie `x[10] = 2`, wówczas wartość $y[10]$ również otrzyma wartość 2. ■

Okazuje się więc, że zrozumienie aliasowania i mechanizmów, które je tworzą, jest kluczowe, jeśli kompilator ma zoptymalizować program. Jak zobaczymy, począwszy od rozdziału 9, istnieje wiele sytuacji, w których będziemy mogli zoptymalizować kod tylko wtedy, gdy będziemy mieć pewność, że określone zmienne nie są aliasowane. Na przykład możemy ustalić, że `x = 2` jest jedynym miejscem, w którym zmienna x otrzymuje jakąś wartość. Jeśli tak jest, możemy zastąpić użycie zmiennej x użyciem stałej 2; na przykład możemy zastąpić przypisanie `a = x + 3` prostszym `a = 5`. Przypuśćmy jednak, że jest tu jeszcze jedna zmienna y, która jest aliasem x. Wówczas przypisanie `y = 4` może mieć nieoczekiwany efekt zmieniający wartość x. Może to również oznaczać, że zastąpienie `a = x + 3` przez `a = 5` było pomyłką; poprawną wartością w tym miejscu może być bowiem 7.

1.6.8. Ćwiczenia do podrozdziału 1.6

Ćwiczenie 1.6.1: Dla strukturalnego bloku kodu w języku C pokazanego na rysunku 1.13(a) podaj wartości przypisane do w, x, y i z.

Ćwiczenie 1.6.2: Powtórz ćwiczenie 1.6.1 dla kodu z rysunku 1.13(b).

Ćwiczenie 1.6.3: Dla kodu o strukturze blokowej z rysunku 1.14, zakładając zwyczajowy statyczny zasięg deklaracji, podaj zasięg każdej z dwunastu deklaracji.

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Kod do ćwiczenia 1.6.1

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(b) Kod do ćwiczenia 1.6.2

RYSUNEK 1.13: Kod o strukturze blokowej

Ćwiczenie 1.6.4: Co zostanie wypisane przez poniższy kod C?

```
#define a (x+1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n"), a; }
void main() { b(); c(); }
```

```
{   int w, x, y, z;     /* Block B1 */
    {   int x, z;       /* Block B2 */
        {   int w, x;   /* Block B3 */ }
    }
    {   int w, x;       /* Block B4 */
        {   int y, z;   /* Block B5 */ }
    }
}
```

RYSUNEK 1.14: Kod o strukturze blokowej dla ćwiczenia 1.6.3

1.7. Podsumowanie

- *Procesory języka.* Zintegrowane środowisko projektowania oprogramowania zawiera wiele różnego typu procesorów języka, takich jak kompilatory, interpretery, asemblery, konsolidatory, loadery, debugery i profilery.
- *Fazy kompilacji.* Kompilator działa w sekwencji kolejnych faz, z których każda przekształca program źródłowy z jednej reprezentacji pośredniej do następnej.
- *Języki maszynowe i asemblery.* Języki maszynowe były językami programowania pierwszej generacji, po których pojawiły się języki asemblerów. Programowanie w tych językach było pracochłonne i podatne na błędy.
- *Modelowanie w projektowaniu kompilatorów.* Projektowanie kompilatorów jest jednym z tych miejsc, w których teoria ma największy wpływ na praktykę. Modele, które okazały się użyteczne, obejmują automaty, gramatyki, wyrażenia regularne, drzewa i wiele innych.
- *Optymalizacja kodu.* Choć kod nie może zostać naprawdę „zoptymalizowany", nauka o ulepszaniu wydajności kodu jest jednocześnie złożona i bardzo ważna. Jest to główna część badania kompilatorów.
- *Języki wyższych poziomów.* Z upływem czasu języki programowania przejmują stopniowo coraz więcej zadań, które wcześniej pozostawały w gestii programisty, takie jak zarządzanie pamięci, sprawdzanie zgodności typów lub równoległe wykonywanie kodu.
- *Kompilatory a architektura komputerów.* Technologia kompilatorów wpływa na rozwój architektury komputerów i jednocześnie podlega wpływom postępów w zakresie architektury. Wiele nowoczesnych innowacji w architekturze uzależnionych jest od zdolności kompilatorów do wydobycia z programów źródłowych możliwości efektywnego wykorzystania funkcjonalności sprzętu.
- *Niezawodność i bezpieczeństwo oprogramowania.* Te same technologie, które pozwalają kompilatorom na optymalizację kodu, mogą być użyte do rozmaitych zadań analizy programów, rozciągających się od wyszukiwania typowych

bugów w programach po odkrywanie podatności programu na jeden z wielu typów nadużyć wynalezionych przez „hackerów".

- *Reguły zasięgu. Zasięgiem* deklaracji x jest kontekst, w którym użycie nazwy x odnosi się do tej deklaracji. Język używa zasięgu statycznego lub leksykalnego, jeśli możliwe jest ustalenie zasięgu deklaracji jedynie przez przeglądanie programu. W przeciwnym razie język używa zasięgu dynamicznego.
- *Środowisko.* Powiązanie nazw z lokalizacjami w pamięci, a następnie z wartościami można opisać przez pojęcie *środowisk*, które mapują nazwy na lokalizacje w pamięci, oraz *stanów*, które mapują lokalizacje na ich wartości.
- *Struktura blokowa.* Języki, które pozwalają na zagnieżdżanie bloków kodu, mają strukturę blokową. Nazwa x w zagnieżdżonym bloku B znajduje się w zasięgu deklaracji D tej nazwy x w otaczającym bloku, jeśli nie ma innej deklaracji x w jakimś bloku leżącym między blokiem zawierającym D a użyciem nazwy x.
- *Przekazywanie parametrów.* Parametry są przekazywane z procedury wywołującej do wywoływanej przez wartość lub przez referencję. Jeśli wielki obiekt jest przekazywany przez wartość, owe przekazywane wartości są w rzeczywistości referencjami do samego obiektu, co ostatecznie daje przekazanie przez referencję.
- *Aliasowanie.* Gdy parametry są (efektywnie) przekazywane przez referencję, dwa formalne parametry mogą odnosić się do tego samego obiektu. Taka sytuacja powoduje, że zmiana jednej zmiennej może również zmienić drugą.

1.8. Bibliografia

Informacje o rozwoju języków programowania, które zostały utworzone i były w użyciu do roku 1967, w tym Fortran, Algol, Lisp i Simula, zawiera publikacja [7]. Omówienie języków powstałych do roku 1982, w tym C, C++, Pascal i Smalltalk, można znaleźć w [1].

GNU Compiler Collection, gcc, to popularne źródło kompilatorów *open-source* dla języków C, C++, Fortranu, Javy i innych [2]. Phoenix to zestaw narzędzi do budowania kompilatorów udostępniający zintegrowane środowiska do budowania faz analizy programu, generowania kodu i optymalizacji omówionych w tej książce [3].

Osobom poszukującym więcej informacji o koncepcjach języków programowania polecamy pozycje [5, 6]. Szersze omówienie architektury komputerów i jej wpływu na kompilowanie programów zawiera [4].

1. Bergin, T.J. i R.G. Gibson, *History of Programming Languages*, ACM Press, New York 1996.

2. `http://gcc.gnu.org/`.

3. `http://research.microsoft.com/phoenix/default.aspx.`
4. Hennessy, J.L. i D.A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan-Kaufmann, San Francisco, CA, 2004.
5. Scott, M.L., *Programming Language Pragmatics*, wyd. 2, Morgan-Kaufmann, San Francisco, CA, 2006.
6. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1996.
7. Wexelblat, R.L., *History of Programming Languages*, Academic Press, New York 1981.

Rozdział 2

Prosty translator sterowany składnią

Rozdział ten stanowi wprowadzenie do technik kompilacji, których omawianiu poświęcone są rozdziały od 3 do 6 tej książki. Techniki te zilustrujemy, tworząc działający program w języku Java tłumaczący instrukcje reprezentatywnego języka programowania na reprezentację pośrednią w formie kodu trójadresowego. W tym rozdziale nacisk jest położony na przód kompilatora, a w szczególności na analizę leksykalną, analizę składniową i generowanie kodu pośredniego. Rozdziały 7 i 8 pokazują, jak generować rozkazy maszynowe z kodu trójadresowego.

Rozpoczniemy od utworzenia prostego, sterowanego składnią translatora odwzorowującego infiksowe wyrażenia arytmetyczne na równoważne wyrażenia postfiksowe (wyrażenia w odwrotnej notacji polskiej). Następnie będziemy rozszerzać ten translator, aby odwzorować fragmenty kodu, takie jak pokazane na rysunku 2.1, na trójadresowy kod w formie widocznej na rysunku 2.2.

```
{
      int i; int j; float[100] a; float v; float x;

      while ( true ) {
            do i = i+1; while ( a[i] < v );
            do j = j-1; while ( a[j] > v );
            if ( i >= j ) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

RYSUNEK 2.1: Fragment kodu do przetłumaczenia

Działający translator w Javie zawarty jest w dodatku A. Użycie języka Java jest wygodne, ale wybór ten nie jest niczym zasadniczym. W istocie, koncepcje przedstawione w tym rozdziale są wcześniejsze zarówno od języka Java, jak i C.

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

RYSUNEK 2.2: Uproszczony kod pośredni dla fragmentu kodu z rysunku 2.1

2.1. Wprowadzenie

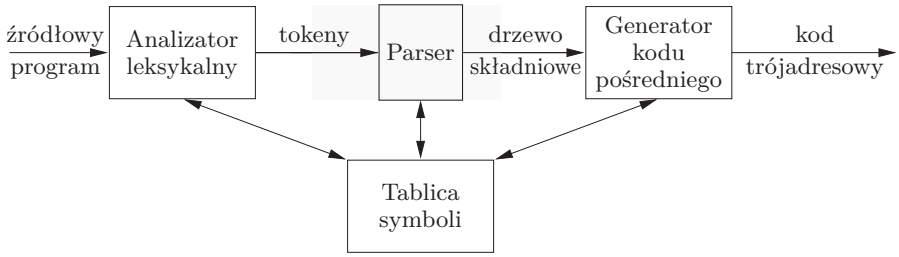
Faza analizy kompilatora dzieli program źródłowy na części składowe i tworzy jego wewnętrzną reprezentację nazywaną kodem pośrednim. Faza syntezy tłumaczy kod pośredni na wynikowy program.

Analiza jest zbudowana wokół „składni" języka, który ma zostać skompilowany. *Składnia* języka programowania opisuje prawidłową formę programów w tym języku, podczas gdy *semantyka* języka definiuje, co te programy znaczą – w tym sensie, co dany program robi, gdy jest wykonywany. Do wyspecyfikowania składni w podrozdziale 2.2 zaprezentujemy szeroko używaną notację nazywaną gramatyką bezkontekstową albo BNF (akronim od Backus-Naur Form). Przy użyciu obecnie dostępnych notacji semantyka języka jest znacznie trudniejsza do opisania niż jego składnia. Z tego względu przy opisie semantyki będziemy używać nieformalnych opisów i pomocnych przykładów.

Oprócz specyfikacji składni języka gramatyka bezkontekstowa może zostać użyta jako pomoc przy tłumaczeniu programów. W podrozdziale 2.3 przedstawimy opierającą się na gramatyce technikę kompilacji znaną jako tłumaczenie sterowane składnią (*syntax-directed translation*). Analizę składniową (*parsing*) zajmiemy się w podrozdziale 2.4.

Reszta tego rozdziału to szybki przegląd modelu przodu kompilatora pokazanego na rysunku 2.3. Zaczniemy od parsera. Na początek rozważymy sterowane składnią tłumaczenie wyrażeń infiksowych na postfiksową formę zapisu, w której operatory następują po swoich operandach. Na przykład postfiksowa forma wyrażenia `9-5+2` to `95-2+`. Tłumaczenie na formę postfiksową jest wystarczająco rozbudowane, aby zilustrować analizę składniową, ale dość proste, aby móc zbudować translator pokazany w całości w podrozdziale 2.5. Ten prosty translator obsługuje wyrażenia takie jak `9-5+2`, złożone z cyfr rozdzielanych znakami plus i minus. Jednym z powodów, dla których zaczynamy od tak prostych wyrażeń,

jest to, że nasz analizator składniowy będzie mógł działać bezpośrednio na pojedynczych znakach dla operatorów i operandów.

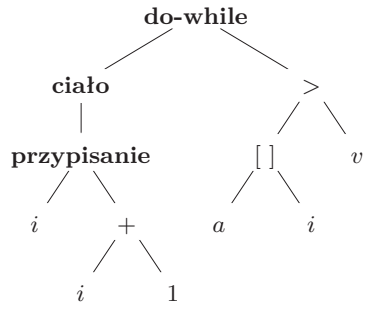


RYSUNEK 2.3: Model przodu kompilatora

Analizator leksykalny pozwala na przetwarzanie przez translator wieloznakowych konstrukcji, takich jak identyfikatory napisane jako sekwencje znaków, ale traktowane jako jednostki nazywane *tokenami* podczas analizy składniowej. Na przykład w wyrażeniu `count + 1` identyfikator `count` jest traktowany jako jednostka. Analizator leksykalny pokazany w podrozdziale 2.6 pozwala na pojawianie się w wyrażeniach (poddawanych analizie składniowej) jednostek (tokenów) takich jak liczby, identyfikatory i „białe znaki" (spacje, tabulatory i znaki nowego wiersza).

Następnie rozważymy generowanie kodu pośredniego. Rysunek 2.4 ilustruje dwie formy kodu pośredniego. Jedna z nich, nazywana *abstrakcyjnym drzewem składniowym* lub po prostu drzewem składniowym, reprezentuje hierarchiczną strukturę składniową programu źródłowego. W modelu przedstawionym na rysunku 2.3 parser tworzy drzewo składniowe, które jest później tłumaczone na kod trójadresowy. Niektóre kompilatory łączą analizę i generowanie kodu pośredniego w jeden komponent.

Korzeń abstrakcyjnego drzewa składniowego pokazanego na rysunku 2.4(a) reprezentuje całą pętlę do-while. Lewa potomna gałąź przedstawia ciało pętli,



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

RYSUNEK 2.4: Kod pośredni dla „`do i = i + 1; while(a[i] < v);`"

składające się jedynie z przypisania `i = i + 1;`. Prawa gałąź reprezentuje warunek `a[i] < v`. Implementacja drzew składniowych zostanie przedstawiona w podrozdziale 2.8.

Inna typowa pośrednia reprezentacja pokazana na rysunku 2.4(b) to sekwencja „trójadresowych" instrukcji; bardziej kompletny przykład widoczny jest na rysunku 2.2. Ta forma kodu pośredniego bierze swą nazwę z instrukcji w postaci $x = y$ **op** z, gdzie **op** jest operatorem binarnym, y i z są adresami operandów, x jest zaś adresem, w którym ma być umieszczony wynik tej operacji. Trójadresowa instrukcja obsługuje co najwyżej jedną operację – typowo obliczenie, porównanie lub rozgałęzienie.

W Dodatku A zbierzemy razem wszystkie techniki przedstawione w tym rozdziale w celu zbudowania front-endu kompilatora w Javie. Front-end tłumaczy instrukcje źródłowe na instrukcje (rozkazy) poziomu asemblera.

2.2. Definiowanie składni

W tym punkcie wprowadzimy notację – „gramatykę bezkontekstową" lub po prostu „gramatykę"– służącą do specyfikowania składni języka. W całej tej książce organizacja front-endów kompilatora oparta jest na gramatykach.

Gramatyka w naturalny sposób opisuje hierarchiczną strukturę większości konstrukcji języków programowania. Na przykład instrukcja if-else w Javie może mieć formę

if (wyrażenie) instrukcja **else** instrukcja

Instrukcja if-else jest złączeniem słowa kluczowego **if**, otwierającego nawiasu, *wyrażenia logicznego*, zamykającego nawiasu, jakiejś *instrukcji*, słowa kluczowego **else** oraz innej instrukcji. Jeśli użyjemy zmiennej *expr* do oznaczenia wyrażenia oraz zmiennej *stmt* do oznaczenia instrukcji, wówczas ta reguła strukturalna może zostać przedstawiona w następujący sposób:

$$stmt \rightarrow \textbf{if} \; (\; expr \;) \; stmt \; \textbf{else} \; stmt$$

W zapisie tym strzałkę należy czytać jako „może mieć postać". Taka reguła nazywana jest produkcją. W produkcji niepodzielne elementy leksykalne, takie jak słowo kluczowe *if* oraz nawiasy, nazywane są symbolami terminalnymi. Zmienne, takie jak *expr* i *stmt*, reprezentują sekwencje symboli terminalnych i są nazywane symbolami nieterminalnymi.

2.2.1. Definicja gramatyki

Gramatykę bezkontekstową budują cztery komponenty:

1. Skończony zbiór *symboli terminalnych* (w skrócie *terminali*), niekiedy określanych mianem „tokenów". Symbole terminalne są elementarnymi symbolami języka definiowanego przez tę gramatykę.

Tokeny *versus* terminale

Analizator leksykalny kompilatora odczytuje znaki programu źródłowego, grupuje je w leksykalnie znaczące jednostki zwane leksemami i produkuje jako wyjście tokeny, reprezentujące te leksemy. Token składa się z dwóch komponentów: nazwy tokenu oraz wartości atrybutu. Nazwa tokenu jest symbolem abstrakcyjnym używanym przez parser podczas analizy składniowej. Często będziemy nazywać te tokeny terminalami, gdyż występują one jako symbole terminalne w gramatyce języka programowania. Wartość atrybutu – o ile jest obecna – jest wskaźnikiem do tablicy symboli zawierającej dodatkowe informacje o tokenie. Te dodatkowe informacje nie są częścią gramatyki, zatem w naszej dyskusji na temat analizy składniowej będziemy często zamiennie używać terminów token i terminal (symbol terminalny).

2. Skończony zbiór *symboli nieterminalnych* (w skrócie *nieterminali*), nazywanych „zmiennymi syntaktycznymi". Każdy symbol nieterminalny oznacza zbiór ciągów utworzonych z symboli terminalnych w sposób, który zamierzamy opisać.
3. Zbiór *produkcji*, przy czym każda produkcja składa się z symbolu nieterminalnego nazywanego nagłówkiem lub lewą stroną produkcji, strzałki oraz sekwencji terminali lub nieterminali, nazywanej ciałem lub prawą stroną produkcji. Intuicyjnym celem produkcji jest określenie jednej z możliwych form konstrukcji językowej; jeśli nagłówkowy nieterminal reprezentuje pewną konstrukcję, wówczas ciało produkcji przedstawia formę zapisu tej konstrukcji.
4. Jeden wyróżniony symbol nieterminalny jest traktowany jako *symbol startowy*.

Specyfikowanie gramatyki wykonujemy, wyliczając jej produkcje, zaczynając od produkcji dla symbolu startowego. Przyjmujemy, że cyfry, znaki specjalne, na przykład `+`, `<` lub `<=` oraz słowa wytłuszczone, takie jak **while**, są symbolami terminalnymi. Nazwa pisana kursywą jest nieterminalem, w przypadku zaś dowolnej nazwy lub symbolu niewyróżnionego kursywą możemy przypuszczać, że jest to terminal[1]. Dla czytelniejszego zapisu będziemy grupować produkcje mające jako nagłówek ten sam nieterminal, po prawej stronie tworząc alternatywę ciał poszczególnych produkcji. Ciała te rozdzielamy symbolem |, który odczytujemy jako „lub".

[1] Indywidualnych liter pisanych kursywą będziemy używać w dodatkowych celach, szczególnie podczas pogłębionej analizy gramatyk w rozdziale 4. Przykładowo będziemy używać X, Y oraz Z przy omawianiu symbolu, który jest albo terminalem, albo nieterminalem. Niemniej jednak dowolna pisana kursywą nazwa zawierająca dwa i więcej znaków nadal będzie reprezentować nieterminal.

Przykład 2.1: Wiele przykładów w tym rozdziale używa wyrażeń złożonych z cyfr oraz znaków plus i minus, na przykład ciągi `9-5+2`, `3-1` lub `7`. Ponieważ znaki plus lub minus muszą występować między dwiema cyframi, będziemy odnosić się do takich wyrażeń jako „list cyfr rozdzielanych znakami plus lub minus". Poniższa gramatyka opisuje składnię takich wyrażeń. Reguły produkcji to:

$$\textit{list} \rightarrow \textit{list}\ \texttt{+}\ \textit{digit} \tag{2.1}$$
$$\textit{list} \rightarrow \textit{list}\ \texttt{-}\ \textit{digit} \tag{2.2}$$
$$\textit{list} \rightarrow \textit{digit} \tag{2.3}$$
$$\textit{digit} \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \tag{2.4}$$

Ciała trzech produkcji z nieterminalnymi listami jako nagłówkami można zgrupować w równoważny zapis:

$$\textit{list} \rightarrow \textit{list}\ \texttt{+}\ \textit{digit} \mid \textit{list}\ \texttt{-}\ \textit{digit} \mid \textit{digit}$$

Zgodnie z naszą konwencją, terminalami tej gramatyki są symbole

`+ - 0 1 2 3 4 5 6 7 8 9`

Nieterminale to pisane kursywą nazwy *list* oraz *digit*, przy czym *list* jest symbolem startowym, gdyż jej produkcja została podana jako pierwsza. ■

Będziemy mówić, że produkcja jest produkcją dla danego nieterminala, jeśli nieterminal ten stanowi nagłówek (lewą stronę) tej produkcji. Ciąg terminali jest sekwencją zero lub więcej terminali. Ciąg zawierający zero terminali, zapisany jako ϵ, nazywany jest pustym ciągiem[2].

2.2.2. Wyprowadzenia

Gramatyka wyprowadza ciągi symboli, zaczynając od symbolu startowego i kolejno zastępując symbol nieterminalny ciałem produkcji dla tego nieterminala. Wszystkie ciągi symboli terminalnych, które mogą zostać wyprowadzone z symbolu startowego, tworzą język zdefiniowany przez tę gramatykę.

Przykład 2.2: Język zdefiniowany przez gramatykę z przykładu 2.1 składa się z list cyfr porozdzielanych znakami plus oraz minus. Dziesięć produkcji dla nieterminalnego symbolu *digit* pozwala zastąpić ten nieterminal dowolnym z terminali od 0 do 9. Zgodnie z produkcją (2.3) pojedyncza cyfra sama z siebie jest listą. Produkcje (2.1) oraz (2.2) wyrażają zasadę, że dowolna lista uzupełniona o znak plus lub minus oraz kolejną cyfrę tworzy nową listę.

Produkcje od (2.1) do (2.4) to wszystko, czego potrzebujemy, aby zdefiniować pożądany język. Na przykład możemy wydedukować, że `9-5+2` jest listą, na podstawie następującego rozumowania:

(a) `9` jest *list* na podstawie produkcji (2.3), gdyż 9 jest *digit*.

[2] Technicznie rzecz ujmując, ϵ może być ciągiem o długości zero symboli z dowolnego alfabetu (zbioru symboli).

(b) `9-5` jest *list* na podstawie produkcji (2.2), gdyż `9` jest *list* i `5` jest *digit*.

(c) `9-5+2` jest *list* na podstawie produkcji (2.1), gdyż `9-5` jest *list* i `2` jest *digit*. ■

Przykład 2.3: Nieco innym rodzajem listy jest lista parametrów w wywołaniu funkcji. W języku Java parametry są ujęte w nawiasach, jak w wywołaniu `max(x,y)` funkcji `max` z parametrami `x` i `y`. Szczególnym przypadkiem takiej listy jest to, że między terminalami `(` i `)` może znajdować się pusta lista parametrów. Możemy zacząć budować gramatykę dla takich sekwencji przy użyciu poniższych produkcji:

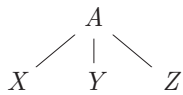
$$\begin{array}{rcl} \textit{call} & \rightarrow & \textbf{id}\ \texttt{(}\ \textit{optparams}\ \texttt{)} \\ \textit{optparams} & \rightarrow & \textit{params} \mid \epsilon \\ \textit{params} & \rightarrow & \textit{params}\ \texttt{,}\ \textit{param} \mid \textit{param} \end{array}$$

Zwróćmy uwagę, że drugie możliwe ciało dla *optparams* („optional parameter list", opcjonalna lista parametrów) to ϵ, oznaczający pusty ciąg symboli. Właśnie tak, *optparams* może zostać zastąpione pustym ciągiem, zatem wywołanie może składać się z nazwy funkcji, po której następuje ciąg dwóch symboli terminalnych `()`. Zauważmy, że produkcje dla *params* są analogiczne dla *list* z przykładu 2.1, przy czym przecinek zastępuje operator arytmetyczny `+` lub `-` i *param* zastępuje *digit*. Nie pokazujemy tu produkcji dla *param*, gdyż parametry są naprawdę arbitralnymi wyrażeniami. Wkrótce omówimy odpowiednie produkcje dla różnych konstrukcji języka, takich jak wyrażenia, instrukcje i tak dalej. ■

Analiza składniowa (*parsing*) polega na pobraniu ciągu symboli terminalnych i ustaleniu, w jaki sposób jest on wyprowadzany z symbolu startowego gramatyki, a jeśli nie może on zostać wyprowadzony z symbolu startowego, zgłaszany jest błąd składniowy w danym ciągu. Parsing jest jednym najbardziej fundamentalnych problemów w całej kompilacji; główne podejścia zostaną omówione w rozdziale 4. W tym rozdziale dla prostoty zaczniemy od programów źródłowych takich jak `9-5+2`, w których każdy znak jest symbolem terminalnym. W ogólnym przypadku, program źródłowy będzie zawierał wieloznakowe leksemy grupowane przez analizator leksykalny w tokeny, których pierwszymi komponentami są terminale przetwarzane przez parser.

2.2.3. Drzewa rozbioru

Drzewo rozbioru (*parse tree*) graficznie pokazuje, jak z symbolu startowego gramatyki wywodzi się ciąg występujący w języku. Jeśli nieterminalne A ma produkcję $A \rightarrow XYZ$, wówczas drzewo analizy może zawierać pośredni węzeł oznaczony etykietą A z trzema potomnymi węzłami opisanymi jako X, Y oraz Z, od lewej do prawej:



Terminologia drzew

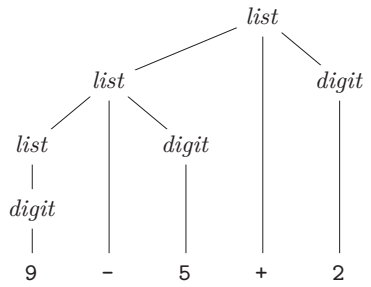
Struktury danych typu *drzewo* licznie pojawiają się w zagadnieniach dotyczących kompilatorów, zatem warto uściślić terminologię.

- Drzewo składa się z jednego lub więcej *węzłów*. Węzły mogą mieć etykiety, którymi w tej książce typowo będą symbole gramatyki. Przy rysowaniu drzewa często przedstawiamy węzły jedynie przez te etykiety.
- Dokładnie jeden węzeł jest korzeniem (*root*). Wszystkie węzły poza korzeniem mają unikatowego *rodzica*; korzeń nie ma rodzica. Przy rysowaniu drzewa umieszczamy rodzica węzła ponad tym węzłem i rysujemy krawędź między nimi. Tym samym korzeń jest najwyższym (szczytowym) węzłem.
- Jeśli węzeł N jest rodzicem węzła M, wówczas M jest *dzieckiem* N. Potomków jednego węzła nazywamy *rodzeństwem*. Ma ono porządek, od lewej do prawej, i gdy rysujemy drzewa, rysujemy dzieci ustalonego węzła w taki właśnie sposób.
- Węzeł pozbawiony dzieci nazywamy *liściem*. Inne węzły – te, które mają jedno lub więcej dzieci – to węzły wewnętrzne.
- *Potomkiem* węzła N jest albo sam węzeł N, albo dziecko węzła N, lub też dziecko dziecka węzła N i tak dalej, przez dowolną liczbę poziomów. Powiemy, że węzeł N jest przodkiem węzła M, jeśli M jest potomkiem węzła N.

Mówiąc formalnie, dla danej gramatyki bezkontekstowej drzewo analizy odpowiadające tej gramatyce jest drzewem o następujących właściwościach:

1. Korzeń oznaczony jest symbolem startowym.
2. Każdy liść jest oznaczony symbolem terminalnym lub ϵ.
3. Każdy węzeł wewnętrzny jest oznaczony symbolem nieterminalnym.
4. Jeśli nieterminalne A jest oznaczeniem jakiegoś wewnętrznego węzła, a $X_1, X_2, \ldots, X_n$ są etykietami potomków tego węzła od lewej do prawej, wówczas w gramatyce musi istnieć produkcja $\mathrm{A} \to X_1 X_2 \cdots X_n$. W tym przypadku każdy z zapisów $X_1, X_2, \ldots, X_n$ oznacza symbol, który jest albo terminalny, albo nieterminalny. W szczególnym przypadku, gdy mamy produkcję $A \to \epsilon$, wówczas węzeł z etykietą A może mieć pojedynczego potomka oznaczonego ϵ.

Przykład 2.4: Wyprowadzenie zapisu `9-5+2` z przykładu 2.2 ilustruje drzewo pokazane na rysunku 2.5. Każdy węzeł w tym drzewie jest oznaczony symbolem



RYSUNEK 2.5: Drzewo rozbioru dla wyrażenia `9-5+2`, odpowiadające gramatyce z przykładu 2.1

gramatyki. Węzeł wewnętrzny i jego dzieci odpowiadają produkcji; węzeł wewnętrzny to nagłówek, a węzły potomne to ciało tej produkcji.

Na rysunku 2.5 korzeń jest oznaczony jako *list*, czyli symbolem startowym gramatyki z przykładu 2.1. Dzieci korzenia są oznaczone od lewej do prawej jako *list*, + oraz *digit*. Zauważmy, że

$$list \rightarrow list \texttt{ + } digit$$

jest produkcją z gramatyki zawartej w przykładzie 2.1. Lewe dziecko korzenia jest analogiczne do korzenia, ale z dzieckiem opatrzonym etykietą `-` zamiast `+`. Każdy z trzech węzłów oznaczonych *digit* ma jedno dziecko oznaczone pojedynczą cyfrą. ■

Liście drzewa rozbioru odczytywane od lewej do prawej pozwalają uzyskać ciąg symboli *wygenerowany* czy też *wyprowadzony* z nieterminalnego korzenia drzewa. Na rysunku 2.5 tym wynikiem jest `9-5+2`, dla wygody wszystkie liście zostały umieszczone na tym samym, najniższym poziomie. Trzeba jednak zauważyć, że nie musimy koniecznie wyrównywać liści w ten sposób. Dowolne drzewo narzuca swoim liściom naturalny porządek od lewej do prawej zgodnie z zasadą, że jeśli X i Y są dziećmi tego samego rodzica i X jest na lewo od Y, to każdy potomek X jest na lewo od dowolnego potomka Y.

Inna definicja języka generowanego przez gramatykę mówi, że jest to zbiór ciągów symboli, z których każdy może zostać wygenerowany przez pewne drzewo rozbioru. Proces znajdowania drzewa rozbioru dla podanego ciągu terminali nazywany jest *parsingiem* tego ciągu.

2.2.4. Niejednoznaczność

Trzeba zachować ostrożność, gdy mówimy o strukturze ciągu wyprowadzanego w danej gramatyce. Gramatyka może dopuszczać więcej niż jedno drzewo rozbioru generujące zadany ciąg terminali. Taka gramatyka nazywana jest niejednoznaczną. Aby wykazać, że dana gramatyka jest niejednoznaczna, wystarczy znaleźć ciąg symboli terminalnych, który daje się utworzyć przez więcej niż jedno

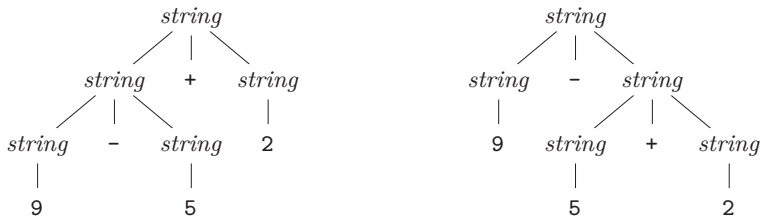
drzewo rozbioru. Ponieważ ciąg z więcej niż jednym drzewem rozbioru zazwyczaj ma również więcej niż jedno znaczenie, wymagane jest projektowanie gramatyk jednoznacznych do celów kompilacji albo używanie gramatyk niejednoznacznych z dodatkowymi regułami rozstrzygającymi niejednoznaczności.

Przykład 2.5: Załóżmy, że użyliśmy pojedynczego nieterminalnego symbolu nazwanego *string* i nie wykonaliśmy rozróżnienia między cyframi (*digit*) i listami (*list*), jak w przykładzie 2.1. Moglibyśmy sformułować taką gramatykę:

$$string \rightarrow string \texttt{+} string \mid string \texttt{-} string \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

Scalenie znaczeń *digit* (cyfra) i *list* (lista) w nieterminalnym symbolu *string* upraszcza sens, gdyż pojedyncza cyfra jest szczególnym przypadkiem listy.

Jednak rysunek 2.6 pokazuje, że wyrażeniu takiemu, jak `9-5+2` odpowiada w tej gramatyce więcej niż jedno drzewo rozbioru. Dwa drzewa dla napisu `9-5+2` odpowiadają dwóm sposobom potencjalnego nawiasowania naszego wyrażenia: `(9-5)+2` oraz `9-(5+2)`. Ten drugi wariant daje wyrażeniu nieoczekiwaną wartość 2 zamiast zwyczajowej wartości 6. Gramatyka z przykładu 2.1 nie pozwala na taką interpretację i gwarantuje jednoznaczną poprawną ewaluację wyrażenia. ■

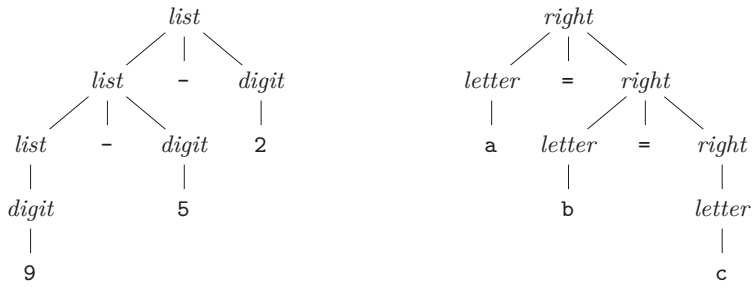


RYSUNEK 2.6: Dwa drzewa rozbioru dla wyrażenia `9-5+2`

2.2.5. Łączność operatorów

Zgodnie z powszechnie używaną konwencją `9+5+2` jest równoważne `(9+5)+2`, a `9-5-2` odpowiada `(9-5)-2`. Gdy operand, taki jak 5 w naszym przykładzie, ma operatory po obu stronach, niezbędna jest konwencja rozstrzygająca, który operator ma zostać zastosowany do tego operandu. Mówimy, że operator `+` ma lewostronną *łączność*, gdyż operand ze znakami plus po obu stronach jest używany do wyliczenia operatora po swojej lewej stronie. W większości języków programowania cztery podstawowe operatory arytmetyczne – dodawanie, odejmowanie, mnożenie i dzielenie są lewostronnie łączne.

Niektóre powszechnie używane operatory, takie jak potęgowanie, są prawostronnie łączne. Również operator przypisania = w języku C i jego pochodne są prawostronnie łączne. Innymi słowy, wyrażenie `a = b = c` jest traktowane tak samo jak wyrażenie `a = (b = c)`.



RYSUNEK 2.7: Drzewa wyprowadzania dla gramatyk z łącznością lewo- i prawostronną

Ciągi takie jak `a = b = c` z operatorem prawostronnie łącznym są generowane przez następującą gramatykę:

$$string \rightarrow string \texttt{ + } string \mid string \texttt{ - } string \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

Różnicę między drzewem wyprowadzania dla operatora lewostronnie łącznego, takiego jak –, a tym dla operatora prawostronnie łącznego, jak =, pokazuje rysunek 2.7. Zauważmy, że drzewo wyprowadzania dla `9-5-2` rośnie w dół w lewą stronę, podczas gdy drzewo dla `a = b = c` rośnie w dół w prawo.

2.2.6. Priorytety operatorów

Rozważmy wyrażenie `9+5*2`. Istnieją dwie możliwe interpretacje tego wyrażenia: `(9+5)*2` lub `9+(5*2)`. Reguły łączności dla `+` i `*` stosują się do występowania tego samego operatora, zatem nie pozwalają one rozstrzygnąć tej niejednoznaczności. Konieczne jest zdefiniowanie reguł określających względne priorytety operatorów, gdy w wyrażeniu występuje więcej niż jeden rodzaj operatora.

Mówimy, że $*$ ma wyższy priorytet niż `+`, jeśli działanie $*$ należy wykonać wcześniej niż działanie `+`. W zwykłej arytmetyce mnożenie i dzielenie mają wyższy priorytet niż dodawanie i odejmowanie. Dlatego argumentem operatora $*$ zarówno w wyrażeniu `9+5*2`, jak i `9*5+2` – będzie 5; wyrażenia te są odpowiednio równoważne napisom `9+(5*2)` oraz `(9*5)+2`.

Przykład 2.6: Gramatyka wyrażeń arytmetycznych może zostać skonstruowana na podstawie tabeli pokazującej łączność i priorytety operatorów. Zacznijmy od czterech podstawowych działań arytmetycznych i tabeli priorytetów pokazującej operatory w kolejności rosnącego priorytetu. Operatory w tym samym wierszu mają taką samą łączność i priorytet:

łączność lewostronna: `+ -`
łączność lewostronna: `* /`

Następnie utworzymy dwa nieterminale *expr* (wyrażenie) oraz *term* (składnik) dla dwóch poziomów priorytetów oraz dodatkowy nieterminal *factor* (czynnik) dla generowania podstawowych jednostek w wyrażeniach. Podstawowymi jednostkami wyrażeń są aktualnie cyfry (*digit*) i wyrażenia (*expr*) w nawiasach.

$$\mathit{factor} \rightarrow \textbf{digit} \mid \texttt{(}\ \mathit{expr}\ \texttt{)}$$

Teraz rozważmy dwuargumentowe operatory $*$ oraz `/`, które mają najwyższy priorytet. Ponieważ operatory te są lewostronnie łączne, produkcje dla nich będą analogiczne do tych dla list, które również były lewostronnie łączne.

$$\begin{array}{lcl} \mathit{term} & \rightarrow & \mathit{term}\ \texttt{*}\ \mathit{factor} \\ & \mid & \mathit{term}\ \texttt{/}\ \mathit{factor} \\ & \mid & \mathit{factor} \end{array}$$

Podobnie *expr* generuje listę składników oddzielanych operatorami addytywnymi.

$$\begin{array}{lcl} \mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{+}\ \mathit{term} \\ & \mid & \mathit{expr}\ \texttt{-}\ \mathit{term} \\ & \mid & \mathit{term} \end{array}$$

Wynikowa gramatyka wygląda więc następująco:

$$\begin{array}{rcl} \mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{+}\ \mathit{term} \mid \mathit{expr}\ \texttt{-}\ \mathit{term} \mid \mathit{term} \\ \mathit{term} & \rightarrow & \mathit{term}\ \texttt{*}\ \mathit{factor} \mid \mathit{term}\ \texttt{/}\ \mathit{factor} \mid \mathit{factor} \\ \mathit{factor} & \rightarrow & \textbf{digit} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \end{array}$$

Przy tej gramatyce wyrażenie jest listą składników oddzielanych znakami `+` lub `-`, składnik jest zaś listą czynników rozdzielanych znakami $*$ lub `/`. Zauważmy, że dowolne wyrażenie ujęte w nawiasy jest czynnikiem, zatem przy użyciu nawiasów możemy budować wyrażenia o dowolnie głębokim zagnieżdżeniu (i dowolnie głębokie drzewa wyprowadzeń). ■

Przykład 2.7: Słowa kluczowe pozwalają nam rozpoznawać instrukcje, gdyż większość instrukcji zaczyna się od pewnego słowa kluczowego lub znaku specjalnego. Wyjątki od tej reguły obejmują przypisania oraz wywołania procedur. Instrukcje zdefiniowane przez (niejednoznaczną) gramatykę pokazaną na rysunku 2.8 są poprawne w Javie.

W pierwszej produkcji dla *stmt* terminalny symbol **id** reprezentuje dowolny identyfikator. Produkcje dla wyrażeń nie są pokazane. Instrukcja przypisania wyspecyfikowana przez pierwszą produkcję jest poprawna w Javie, choć Java traktuje znak `=` jako operator przypisania, który może wystąpić wewnątrz wyrażenia. Na przykład Java pozwala na konstrukcję `a = b = c`, której nie dopuszcza ta gramatyka.

Nieterminalny symbol *stmts* generuje potencjalnie pustą listę instrukcji. Druga produkcja dla *stmts* generuje pustą listę ϵ. Pierwsza produkcja generuje potencjalnie pustą listę instrukcji, po której następuje instrukcja.

Uogólnienie gramatyki wyrażeń z przykładu 2.6

O czynniku możemy myśleć jako o wyrażeniu, które nie może zostać „rozdarte" przez dowolny operator. Mówiąc „rozdarcie", mamy na myśli to, że umieszczenie operatora obok czynnika z dowolnej strony nie spowoduje, że dowolna część czynnika inna niż całość stanie się operandem tego operatora. Jeśli czynnik jest wyrażeniem w nawiasach, nawiasy chronią go przed takim „rozerwaniem", a gdy czynnik jest pojedynczym operandem, nie może zostać rozdzielony.

Składnik (niebędący zarazem czynnikiem) jest wyrażeniem, które może zostać rozdarte przez operatory o wyższym priorytecie: $*$ oraz `/`, ale nie przez operatory o niższym priorytecie. Wyrażenie (niebędące ani składnikiem, ani czynnikiem) może zostać rozdzielone przez dowolny operator.

Możemy uogólnić tę koncepcję na dowolną liczbę n poziomów priorytetów wykonywania działań. Potrzebujemy do tego $n+1$ symboli nieterminalnych. Pierwszy, podobnie jak *factor* z przykładu 2.6, nigdy nie może zostać rozdarty. Typowe ciała reguł produkcji dla takiego nieterminala to jedynie pojedyncze operandy i wyrażenia w nawiasach. Następnie dla każdego kolejnego poziomu priorytetu mamy jeden symbol nieterminalny reprezentujący wyrażenia, które mogą zostać rozdzielone jedynie przez operatory znajdujące się na tym samym poziomie priorytetu lub wyższym. Typowo produkcje dla tych nieterminali mają ciała reprezentujące użycie operatorów z tego poziomu priorytetu plus jedno ciało, które jest pojedynczym nieterminalem z kolejnego wyższego poziomu priorytetu.

Rozmieszczenie średników jest subtelne; pojawiają się one na końcu każdego ciała, które nie kończy się *stmt*. To podejście chroni przed wstawianiem średników po instrukcjach takich jak if- i while-, które kończą się zagnieżdżonymi podinstrukcjami. Gdy zagnieżdżona podinstrukcja jest przypisaniem albo konstrukcją do-while, średnik zostanie wygenerowany jako część podinstrukcji. ■

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{id} = \mathit{expression}\ ; \\
 & | & \textbf{if}\ (\ \mathit{expression}\ )\ \mathit{stmt} \\
 & | & \textbf{if}\ (\ \mathit{expression}\ )\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
 & | & \textbf{while}\ (\ \mathit{expression}\ )\ \mathit{stmt} \\
 & | & \textbf{do}\ \mathit{stmt}\ \textbf{while}\ (\ \mathit{expression}\ )\ ; \\
 & | & \{\ \mathit{stmts}\ \} \\
 & & \\
\mathit{stmts} & \rightarrow & \mathit{stmts}\ \mathit{stmt} \\
 & | & \epsilon
\end{array}
$$

RYSUNEK 2.8: Gramatyka podzbioru instrukcji języka Java

2.2.7. Ćwiczenia do podrozdziału 2.2

Ćwiczenie 2.2.1: Rozważmy gramatykę bezkontekstową

$$S \rightarrow S\ S\ \texttt{+} \mid S\ S\ \texttt{*} \mid \mathbf{a}$$

(a) Pokaż, jak wygenerować ciąg `aa+a*` z tej gramatyki.

(b) Skonstruuj drzewo rozbioru dla tego ciągu.

(c) Jaki język generuje ta gramatyka? Odpowiedź uzasadnij.

Ćwiczenie 2.2.2: Jaki język jest generowany przez poniższe gramatyki? Uzasadnij odpowiedź w każdym przypadku.

(a) $S \rightarrow 0\ S\ 1 \mid 0\ 1$

(b) $S \rightarrow \texttt{+}\ S\ S \mid \texttt{-}\ S\ S \mid \mathbf{a}$

(c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$

(d) $S \rightarrow \mathbf{a}\ S\ \mathbf{b}\ S \mid \mathbf{b}\ S\ \mathbf{a}\ S \mid \epsilon$

(e) $S \rightarrow \mathbf{a} \mid S\ \texttt{+}\ S \mid S\ S \mid S\ \texttt{*} \mid (\ S\ )$

Ćwiczenie 2.2.3: Które z gramatyk w ćwiczeniu 2.2.2 są niejednoznaczne?

Ćwiczenie 2.2.4: Skonstruuj jednoznaczną, bezkontekstową gramatykę dla każdego z poniższych języków. W każdym przypadku wykaż, że proponowana gramatyka jest poprawna.

1. Wyrażenia arytmetyczne w notacji postfiksowej (odwrotnej notacji polskiej).
2. Lewostronnie łączne listy identyfikatorów rozdzielanych przecinkami.
3. Prawostronnie łączne listy identyfikatorów rozdzielanych przecinkami.
4. Wyrażenia arytmetyczne dla liczb całkowitych i identyfikatorów z czterema operatorami dwuargumentowymi `+,-,*,/`.

! e) Dołącz jednoargumentowy plus i minus do operatorów arytmetycznych z punktu (d).

Ćwiczenie 2.2.5: (a) Wykaż, że wszystkie ciągi dwójkowe generowane przez poniższą gramatykę mają wartości podzielne przez 3. *Wskazówka*: użyj indukcji po liczbie węzłów w drzewie rozbioru.

$$num \rightarrow \texttt{11} \mid 1001 \mid num\ \texttt{0} \mid num\ num$$

(b) Czy ta gramatyka generuje wszystkie ciągi dwójkowe, których wartość jest podzielna przez 3?

Ćwiczenie 2.2.6: Skonstruuj bezkontekstową gramatykę do generowania rzymskich cyfr.

2.3. Translacja sterowana składnią

Tłumaczenie sterowane składnią odbywa się przez przypisanie do produkcji w gramatyce reguł lub fragmentów programu. Na przykład rozważmy wyrażenie *expr* generowane przez produkcję

$$expr \rightarrow expr_1 \texttt{ + } term$$

W tym przypadku *expr* jest sumą dwóch podwyrażeń: $expr_1$ oraz *term* (dolny indeks w $expr_1$ został użyty jedynie w celu odróżnienia wystąpienia *expr* w ciele produkcji od nagłówka tej produkcji). Możemy przetłumaczyć *expr*, wykorzystując jego strukturę, jak w poniższym pseudokodzie:

```
przetłumacz expr1;
przetłumacz term;
zajmij_się +;
```

Przy użyciu innego wariantu tego pseudokodu w podrozdziale 2.8 zbudujemy drzewo składniowe dla *expr*, budując drzewa składniowe dla $expr_1$ oraz *term*, a następnie obsługując + przez skonstruowanie dla niego węzła. Dla wygody przykład w tym punkcie będzie się zajmował tłumaczeniem wyrażeń infiksowych do notacji postfiksowej.

W tym podrozdziale przedstawimy dwie koncepcje związane z tłumaczeniem sterowanym składnią:

- *Atrybuty.* Terminem *atrybut* będziemy nazywać dowolną wielkość powiązaną z konstrukcją programową. Przykłady atrybutów to typy danych w wyrażeniach, liczba instrukcji w wygenerowanym kodzie lub lokalizacja pierwszej instrukcji w wygenerowanym kodzie dla tej konstrukcji, aby wymienić tylko niektóre z wielu możliwości. Ponieważ używamy symboli gramatyki (nieterminali i terminali) do reprezentowania konstrukcji programowych, rozszerzamy pojęcie atrybutów z konstrukcji na symbole, które je reprezentują.
- (*Sterowane składnią*) *schematy translacji.* Schemat translacji to zapis przypisujący fragmenty programu do produkcji gramatyki. Fragmenty programu są wykonywane, gdy produkcja jest używana podczas analizy składniowej. Połączony wynik tych wszystkich fragmentarycznych wykonań w kolejności narzuconej przez analizę składniową tworzy tłumaczenie programu, który został poddany procesowi analizy/syntezy.

Translacje sterowane składnią będą używane w dalszej części tego rozdziału do tłumaczenia wyrażeń infiksowych na notację postfiksową, do obliczania wyrażeń i do budowania drzew składniowych dla konstrukcji programowych. Bardziej szczegółowe omówienie formalizmu sterowanego składnią zawiera rozdział 5.

2.3.1. Notacja postfiksowa

Przykłady w tym podrozdziale dotyczą tłumaczenia na notację postfiksową. Notację taką dla wyrażenia E można zdefiniować indukcyjnie, jak poniżej:

1. Jeżeli E jest zmienną lub stałą, wówczas zapisem postfiksowym E jest samo E.
2. Jeżeli E jest wyrażeniem w postaci E_1 **op** E_2, gdzie **op** jest dowolnym operatorem dwuargumentowym, wówczas notacja postfiksowa dla E ma postać $E_1'E_2'$ **op**, gdzie E_1' i E_2' są odpowiednio postfiksowymi postaciami wyrażeń E_1 i E_2.
3. Jeżeli E jest wyrażeniem w nawiasach w postaci (E_1), wówczas notacja postfiksowa dla E jest taka sama jak notacja postfiksowa dla E_1.

Przykład 2.8: Notacja postfiksowa dla `(9-5)+2` to `95-2+`. Translacjami `9`, `5` i `2`, jako stałych, są one same, zgodnie z regułą (1). Następnie translacją `9-5` jest `95-`, zgodnie z regułą (2). Translacją `(9-5)` jest to samo wyrażenie w nawiasie, zgodnie z regułą (3). Po przetłumaczeniu podwyrażenia w nawiasach możemy zastosować regułę (2) do całego wyrażenia, używając `(9-5)` w roli E_1 oraz `2` w roli E_2, aby uzyskać wynik `95-2+`.

Innym przykładem może być notacja postfiksowa dla `9-(5+2)`: `952+-`. W tym przypadku najpierw `5+2` jest tłumaczone na `52+`, po czym to wyrażenie staje się drugim argumentem operatora minus. ■

W notacji postfiksowej nie są potrzebne żadne nawiasy, gdyż pozycja i liczba argumentów operatorów pozwala zdekodować wyrażenie postfiksowe w tylko jeden, właściwy sposób. Cały „trik" polega na powtarzalnym przeglądaniu ciągu postfiksowego od lewej, aż natrafimy na operator. Następnie sięgamy w lewo po odpowiednią liczbę operandów i grupujemy ten operator z jego operandami. Po obliczeniu wyniku działania operatora na jego operandach zastępujemy całą grupę (wykorzystane operandy i operator) wynikiem, po czym powtarzamy proces od początku, przeglądając ciąg w prawo i szukając kolejnego operatora.

Przykład 2.9: Rozważmy wyrażenie postfiksowe `952+-3*`. Przeglądając je od lewej, najpierw natrafiamy na znak plus. Sięgając w lewo od niego, znajdujemy operandy `5` i `2`. Ich suma zastępuje człon `52+` i otrzymujemy ciąg `97-3*`. Teraz „najbardziej lewym" operatorem jest znak minus, a jego operandy to `9` i `7`. Zastępując je wynikiem odejmowania, otrzymujemy `23*`. Ostatni znak mnożenia stosowany jest zatem do liczb `2` i `3`, dając rezultat `6`. ■

2.3.2. Syntetyzowane atrybuty

Koncepcja powiązania pewnych wielkości z konstrukcjami programowymi – na przykład wartości liczbowych i typów z wyrażeniami – może zostać wyrażona w terminach gramatyki. Do symboli nieterminalnych i terminalnych przypisujemy atrybuty. Następnie dołączamy reguły do produkcji tej gramatyki; reguły te

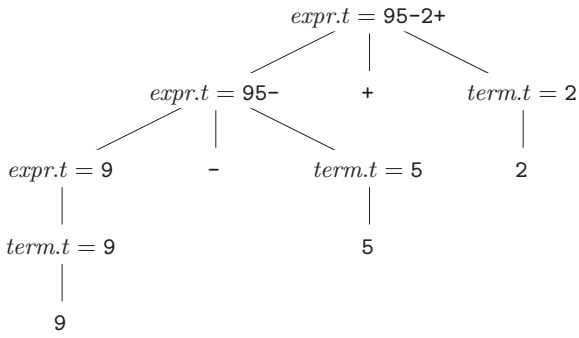
opisują, jak atrybuty są obliczane dla tych węzłów drzewa rozbioru, w których użyte są te produkcje dla powiązania węzła z jego dziećmi.

Definicja sterowana składnią określa powiązanie

1. Każdego symbolu gramatycznego ze zbiorem jego atrybutów oraz
2. Każdej produkcji ze zbiorem reguł semantycznych określających sposób obliczania wartości atrybutów powiązanych z symbolami występującymi w danej produkcji.

Atrybuty mogą być obliczane zgodnie z następującym algorytmem: dla danego ciągu wejściowego x konstruowane jest drzewo rozbioru dla x. Następnie stosowane są reguły semantyczne w celu obliczenia atrybutów dla każdego węzła w drzewie rozbioru, jak poniżej.

Przyjmijmy, że węzeł N w drzewie rozbioru etykietowany jest symbolem gramatycznym X. Piszemy $X.a$, aby oznaczyć wartość atrybutu a dla X w tym węźle. Drzewo rozbioru pokazujące wartości atrybutów dla każdego węzła nazywane jest *dekorowanym drzewem rozbioru* (*annotated parse tree*). Na przykład rysunek 2.9 pokazuje dekorowane drzewo rozbioru dla wyrażenia `9-5+2` z atrybutem t powiązanym z nieterminalnymi symbolami *expr* i *term*. Wartość `95-2+` atrybutu dla węzła korzenia jest zapisem postfiksowym dla wyrażenia `9-5+2`. Wkrótce pokażemy, jak te wyrażenia są obliczane.



RYSUNEK 2.9: Wartości atrybutów dla węzłów drzewa rozbioru

Atrybut nazywamy *syntetyzowanym*, gdy jego wartość w węźle N drzewa rozbioru jest determinowana przez wartości atrybutów dla dzieci węzła N i samego węzła N. Syntetyzowane atrybuty mają tę pożądaną właściwość, że mogą zostać wyliczone przez pojedyncze przejście drzewa rozbioru od dołu do góry (*bottom-up*). W podrozdziale 5.1.1 omówimy inny ważny rodzaj atrybutu: atrybut „dziedziczony". Mówiąc nieformalnie, dziedziczone atrybuty mają wartości w pewnym węźle drzewa rozbioru wynikające z wartości atrybutów dla samego węzła, jego rodzica oraz jego rodzeństwa w drzewie.

Przykład 2.10: Etykietowane drzewo rozbioru pokazane na rysunku 2.9 oparte jest na definicji sterowanej składnią z rysunku 2.10, opisującej tłumaczenie wyrażeń złożonych z cyfr oddzielanych znakami plus lub minus na notację postfiksową.

Każdy nieterminal ma atrybut t, którego wartością jest ciąg reprezentujący notację postfiksową dla wyrażenia wygenerowanego przez ten nieterminal w drzewie rozbioru. Symbol ||w regule semantycznej jest operatorem złączenia (konkatenacji) ciągów.

PRODUKCJA	REGUŁY SEMANTYCZNE
$expr \rightarrow expr_1$ + $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1$ - $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow$ 0	$term.t = '0'$
$term \rightarrow$ 1	$term.t = '1'$
...	...
$term \rightarrow$ 9	$term.t = '9'$

RYSUNEK 2.10: Sterowana składnią definicja dla translacji notacji infiksowej na postfiksową

Postfiksową formą cyfry jest sama cyfra, czyli reguła semantyczna powiązana z produkcją $term \rightarrow$ 9 definiuje, że $term.t$ ma być samą cyfrą 9, ilekroć produkcja ta zostanie użyta w węźle drzewa rozbioru. Inne cyfry są tłumaczone analogicznie. Jako inny przykład, gdy zastosowana jest produkcja $expr \rightarrow term$, wartość $term.t$ staje się wartością $expr.t$.

Produkcja $expr \rightarrow expr_1$+ $term$ wyprowadza wyrażenie zawierające operator plus[3]. Lewy operand operatora plus jest podany jako $expr_1$, a prawy operand to $term$. Reguła semantyczna

$$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$$

powiązana z tą produkcją konstruuje wartość atrybutu $expr.t$, złączając formy postfiksowe $expr_1.t$ oraz $term.t$ lewego i prawego operandu, po czym dołącza do nich znak plus. Reguła ta jest sformalizowanym zapisem definicji „wyrażenia postfiksowego". ■

2.3.3. Proste definicje sterowane składnią

Sterowana składnią definicja przedstawiona w przykładzie 2.10 ma następującą ważną właściwość: ciąg reprezentujący tłumaczenie symbolu nieterminalnego w nagłówku każdej produkcji jest złączeniem tłumaczeń nieterminali zawartych w ciele produkcji w tej samej kolejności, w jakiej występują one w produkcji, prze-

[3] W tej i wielu innych regułach ten sam symbol nieterminalny (w tym przypadku $expr$) pojawia się kilka razy. Celem użycia indeksu 1 w $expr_1$ jest rozróżnienie dwóch wystąpień $expr$ w produkcji; „1" nie jest częścią symbolu nieterminalnego. Więcej szczegółów zawiera ramka „Konwencje rozróżniania użycia symbolu nieterminalnego".

Konwencje rozróżniania użycia symbolu nieterminalnego

W regułach często zachodzi potrzeba rozróżnienia wielokrotnego użycia tego samego nieterminalnego symbolu w nagłówku oraz w ciele produkcji, jak to ma miejsce w przykładzie 2.10. Przyczyną jest to, że w drzewie rozbioru różne węzły etykietowane tym samym nieterminalem zazwyczaj mają różne wartości atrybutów podczas tłumaczenia. Z tego względu stosujemy następującą konwencję: symbol nieterminalny pojawia się bez indeksu w nagłówku reguły produkcji oraz z różnymi indeksami dolnymi w ciele reguły. Wszystkie one są wystąpieniami tego samego nieterminala i indeks nie jest częścią jego nazwy. Jednak dzięki temu czytelnik będzie mógł zauważyć różnice między przykładami określonych translacji, gdzie ta konwencja jest używana, w ogólnej zaś postaci zapisu produkcji $A \to X_1 X_2, \ldots, X_n$ indeksowane X reprezentują pewną listę symboli gramatycznych i *nie* są wystąpieniami jednego określonego symbolu nieterminalnego o nazwie X.

platanymi pewnymi opcjonalnymi, dodatkowymi ciągami. Definicja sterowana składnią o tej właściwości nazywana jest *prostą* (*simple*).

Przykład 2.11: Rozważmy pierwszą produkcję i regułę semantyczną z rysunku 2.10:

PRODUKCJA	REGUŁY SEMANTYCZNE
$expr \to expr_1$ + $term$	$expr.t \;=\; expr_1.t \;\|\|\; term.t \;\|\|\; '+'$

(2.5)

W tym przypadku translacja $expr.t$ jest złączeniem translacji wyrażeń $expr_1$ oraz $term$, uzupełnionym symbolem +. Zauważmy, że $expr_1$ i $term$ występują w tej samej kolejności w ciele produkcji i w regule semantycznej. Nie ma tu dodatkowych symboli przed lub między ich tłumaczeniami. W tym przykładzie jedyny dodatkowy symbol pojawia się na końcu. ■

Przy omawianiu schematów translacji zauważymy, że prosta definicja sterowana składniowo może zostać zaimplementowana przez wypisywanie tylko dodatkowych ciągów w tej kolejności, w jakiej występują w definicji.

2.3.4. Przechodzenie drzewa

Terminu *przechodzenie drzewa* będziemy używać do opisywania obliczania atrybutów i specyfikowania wykonywania fragmentów kodu w schemacie translacji. Przechodzenie drzewa zaczyna się od korzenia i wymaga odwiedzenia wszystkich węzłów drzewa w pewnej kolejności.

Przechodzenie *drzewa w głąb* (*depth-first*) rozpoczyna się od korzenia, po czym rekurencyjnie odwiedzamy dzieci każdego węzła w dowolnej kolejności,

Przechodzenie *preorder* i *postorder*

Przechodzenie *preorder* i *postorder* to dwa ważne szczególne przypadki przechodzenia drzewa w głąb, w których odwiedzamy dzieci każdego węzła w kolejności od lewej do prawej.

Przechodzenie drzewa często jest wykonywane w celu realizacji jakiegoś określonego działania w każdym węźle. Jeśli działanie to jest wykonywane, gdy po raz pierwszy odwiedzamy dany węzeł, to tego typu przechodzenie nazywamy *preorder*. Analogicznie, jeśli działanie wykonywane jest bezpośrednio przed opuszczeniem węzła po raz ostatni, przechodzenie drzewa nazywamy *postorder*. Procedura *visit*(N) na rysunku 2.11 jest przykładem przechodzenia postorder.

Przechodzenie preorder i postorder definiują odpowiadające im uporządkowanie węzłów na podstawie tego, kiedy działanie w danym węźle jest wykonywane. Uporządkowanie preorder (pod)drzewa z korzeniem w węźle N składa się z N, po którym następują uporządkowania preorder poddrzew dla każdego z dzieci tego węzła (jeśli istnieją), od lewej do prawej. Uporządkowanie postorder (pod)drzewa z korzeniem w N składa się z uporządkowań postorder każdego poddrzewa dla dzieci N (jeśli istnieją), ponownie od lewej do prawej, po czym następuje sam węzeł N.

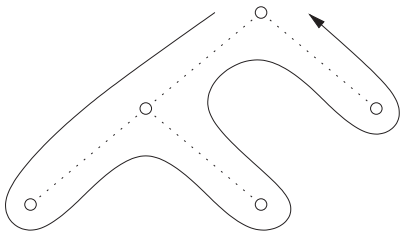
niekoniecznie od lewej do prawej. Nazwa wywodzi się stąd, że w tym procesie odwiedzamy nieodwiedzone jeszcze dziecko węzła, gdy tylko jest to możliwe, zatem docieramy do węzłów oddalonych od korzenia („zagłębionych") tak szybko, jak się da.

Procedura *visit*(N) pokazana na rysunku 2.11 jest przechodzeniem najpierw w głąb, która odwiedza dzieci węzła w kolejności od lewej do prawej, co pokazuje rysunek 2.12. W tym przechodzeniu uwzględniliśmy przetwarzanie reguł translacji w każdym węźle bezpośrednio przed zakończeniem pracy z tym węzłem (czyli gdy przetwarzanie reguł translacji wszystkich dzieci zostało na pewno wykonane). W ogólności działania powiązane z przechodzeniem drzewa mogą być dowolną wybraną operacją lub w ogóle niczym.

Definicja sterowana składniowa nie narzuca żadnego określonego porządku obliczania atrybutów w drzewie rozbioru. Akceptowalna jest dowolna kolejność

```
procedure visit(node N) {
    for ( wszystkie dzieci N, od lewej do prawej ) {
        visit(C);
    }
    przetworzenie reguł semantycznych w węźle N
}
```

RYSUNEK 2.11: Przechodzenie drzewa w głąb



RYSUNEK 2.12: Przykład przechodzenia drzewa w głąb

obliczania, która pozwoli uzyskać uzyskać atrybut α po znalezieniu wszystkich innych atrybutów, od których α jest zależne. Syntetyzowane atrybuty mogą zostać wyliczone podczas dowolnego przejścia z dołu do góry, czyli takiego, które wylicza atrybuty węzła po obliczeniu atrybutów jego dzieci. W ogólności, w przypadku obecności zarówno atrybutów syntetyzowanych, jak i dziedziczonych, zagadnienie kolejności ich wyliczania jest dość złożone. Więcej informacji na ten temat zawiera podrozdział 5.2.

2.3.5. Schematy translacji

Definicja sterowana składnią z rysunku 2.10 tworzy tłumaczenie przez dołączanie do węzłów w drzewie rozbioru atrybutów będących ciągami znaków. Rozważymy teraz alternatywne podejście, które nie wymaga manipulowania ciągami; tworzy ono to samo tłumaczenie stopniowo, przez wykonywanie fragmentów programu.

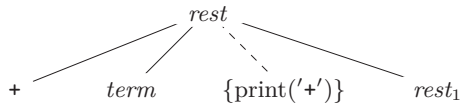
Schematy translacji sterowane składnią to notacja opisująca tłumaczenie przez przypisanie fragmentów programu do produkcji gramatyki. Schemat translacji jest podobny do definicji sterowanej składnią z wyjątkiem tego, że kolejność przetwarzania reguł semantycznych jest jawnie określona.

Fragmenty programu osadzone w ciałach produkcji nazywane są akcjami semantycznymi. Pozycja, w której dana akcja ma zostać wykonana, jest wskazywana przez obramowanie jej nawiasami klamrowymi i umieszczenie w ciele produkcji, jak poniżej

$$rest \;\rightarrow\; + \; term \;\; \{\text{print}('+')\} \;\; rest_1$$

Powinniśmy zobaczyć reguły tego typu, gdy będziemy rozważać alternatywną formę gramatyki dla wyrażeń, w której symbol nieterminalny *rest* reprezentuje „wszystko z wyjątkiem pierwszego składnika wyrażenia". Taka forma gramatyki zostanie omówiona w podrozdziale 2.4.5. Ponownie indeks w $rest_1$ służy odróżnieniu tego wystąpienia nieterminala *rest* w ciele produkcji od wystąpienia *rest* w nagłówku.

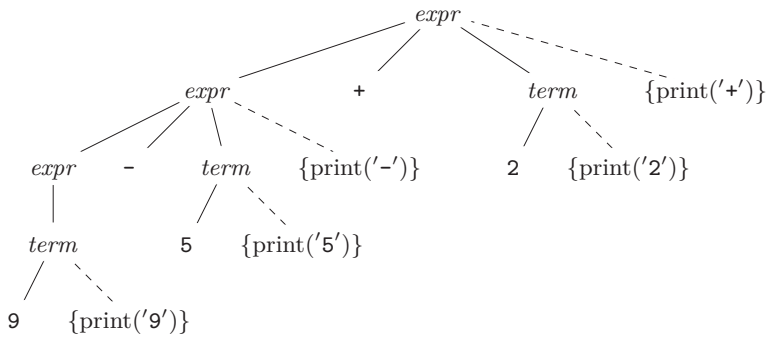
Przy rysowaniu drzewa rozbioru dla schematu transakcji zaznaczamy akcję przez utworzenie dodatkowego dziecka, połączonego linią przerywaną z węzłem odpowiadającym nagłówkowi produkcji. Na przykład część drzewa rozbioru dla powyższej produkcji i akcji pokazana jest na rysunku 2.13. Węzeł dla akcji



RYSUNEK 2.13: Dodatkowy liść tworzony dla akcji semantycznej

semantycznej nie ma dzieci, zatem akcja ta jest wykonywana, gdy węzeł zostanie odwiedzony po raz pierwszy.

Przykład 2.12: Drzewo rozbioru na rysunku 2.14 zawiera instrukcje `print` w dodatkowych liściach, które są połączone przerywanymi liniami z wewnętrznymi węzłami drzewa. Schemat translacji pokazany jest na rysunku 2.15. Leżąca w tle gramatyka generuje wyrażenia złożone z cyfr rozdzielanych znakami plus i minus. Akcje osadzone w ciałach produkcji tłumaczą te wyrażenia na notację postfiksową przy założeniu, że wykonamy przejście przez drzewo w głąb od lewej do prawej i wykonamy każdą instrukcję `print` przy odwiedzeniu odpowiedniego liścia.



RYSUNEK 2.14: Akcje tłumaczące `9-5+2` na `95-2+`

expr	→	*expr*$_1$ `+` *term*	{print('+')}
expr	→	*expr*$_1$ `-` *term*	{print('-')}
expr	→	*term*	
term	→	`0`	{print('0')}
term	→	`1`	{print('1')}
		...	
term	→	`9`	{print('9')}

RYSUNEK 2.15: Akcje tłumaczenia na notację postfiksową

Korzeń na rysunku 2.14 reprezentuje pierwszą produkcję z rysunku 2.15. Przy przechodzeniu drzewa w kolejności postorder najpierw wykonujemy wszystkie akcje w lewym poddrzewie korzenia dla lewego operandu, który podobnie jak korzeń również jest opatrzony etykietą *expr*. Następnie odwiedzamy liść +, w którym nie ma żadnej akcji. Z kolei wykonujemy akcje w poddrzewie dla prawego operandu *term*, a na koniec akcję semantyczną { print('+') } w dodatkowym węźle.

Ponieważ produkcje dla *term* zawierają tylko cyfry po prawej stronie, odpowiednia cyfra jest wypisywana przez akcje dla tych produkcji. Żadne wyjście nie jest potrzebne dla produkcji $expr \rightarrow term$ i operator musi być wypisany tylko w akcjach dla każdej z dwóch pierwszych produkcji. Gdy wszystkie akcje z rysunku 2.14 zostaną wykonane podczas przechodzenia drzewa rozbioru w kolejności postorder, to skutkiem tego będzie wypisanie `95-2+`. ■

Zauważmy, że choć schematy z rysunków 2.10 i 2.15 produkują to same tłumaczenie, jest ono konstruowane w odmienny sposób. Schemat 2.10dołącza ciągi znaków jako atrybuty do węzłów drzewa, podczas gdy schemat 2.15 wypisuje tłumaczenie stopniowo przez akcje semantyczne.

Akcje semantyczne w drzewie pokazanym na rysunku 2.14 tłumaczą wyrażenie infiksowe `9-5+2` na `95-2+`, wypisując każdy znak z oryginalnego wyrażenia dokładnie jeden raz, bez konieczności używania jakiejś formy pamięci na tłumaczenie podwyrażeń. Gdy wyjście tworzone jest przyrostowo w opisany sposób, istotna jest kolejność, w jakiej znaki są wypisywane.

Implementacja schematu translacji musi zagwarantować, że akcje semantyczne będą wykonywane w kolejności, w jakiej występują podczas przechodzenia drzewa rozbioru w kolejności postorder. Implementacja ta nie musi faktycznie konstruować drzewa (i często nie robi tego), jeśli zagwarantuje, że akcje semantyczne są wykonywane w takiej kolejności, jakbyśmy skonstruowali drzewo rozbioru i wykonali akcje podczas przechodzenia postorder tego drzewa.

2.3.6. Ćwiczenia do podrozdziału 2.3

Ćwiczenie 2.3.1: Skonstruuj sterowany składnią schemat translacji, który tłumaczy wyrażenia arytmetyczne z notacji infiksowej na notację prefiksową, w której operator występuje przed operandami; na przykład $-xy$ jest notacją prefiksową dla $x - y$. Zbuduj etykietowane drzewa rozbioru dla danych wejściowych `9-5+2` oraz `9-5*2`.

Ćwiczenie 2.3.2: Skonstruuj sterowany składnią schemat translacji, który tłumaczy wyrażenia arytmetyczne z notacji postfiksowej na infiksową. Zbuduj etykietowane drzewa rozbioru dla danych wejściowych `95-2*` oraz `952*-`.

Ćwiczenie 2.3.3: Skonstruuj sterowany składnią schemat translacji, który tłumaczy liczby całkowite na cyfry rzymskie.

Ćwiczenie 2.3.4: Skonstruuj sterowany składnią schemat translacji, który tłumaczy cyfry rzymskie do 2000 na liczby całkowite.

Ćwiczenie 2.3.5: Skonstruuj sterowany składnią schemat translacji, który tłumaczy postfiksowe wyrażenia arytmetyczne na równoważne wyrażenia prefiksowe.

2.4. Analiza składniowa

Analiza składniowa (*parsing*) to proces ustalania, czy i jak dany ciąg symboli terminalnych może zostać wygenerowany przez gramatykę. W omawianiu tego problemu pomocne będzie myślenie o budowaniu drzewa rozbioru, nawet jeśli w praktyce kompilator może takowego nie konstruować. W każdym przypadku jednak analizator składniowy (*parser*) zasadniczo musi być w stanie skonstruować drzewo; w przeciwnym razie nie będzie można zagwarantować poprawności tłumaczenia.

W tym podrozdziale wprowadzimy metodę nazywaną „metodą zejść rekurencyjnych" (*recursive descent*), która może być używana zarówno do analizy składniowej, jak i implementacji translatorów sterowanych składnią. Pełny program w Javie implementujący schemat translacji z rysunku 2.15 pokażemy w kolejnym punkcie. Sensowną alternatywą jest użycie narzędzia programowego do wygenerowania translatora bezpośrednio ze schematu translacji. W podrozdziale 4.9 opiszemy takie narzędzie – Yacc. Umożliwia ono implementację schematu translacji z rysunku 2.15 bez modyfikacji.

Dla dowolnej gramatyki bezkontekstowej istnieje parser, który wymaga czasu rzędu $O(n^3)$ do przetworzenia ciągu zawierającego n terminali. Jednak czas proporcjonalny do sześcianu długości wejścia jest w ogólności zbyt kosztowny. Szczęśliwie dla rzeczywistych języków programowania możemy zazwyczaj zaprojektować gramatykę, która może być szybko przeanalizowana. Algorytmy o czasowej złożoności liniowej wystarczają do analizy zasadniczo wszystkich języków, jakie pojawiają się w praktyce. Parsery języków programowania niemal zawsze wykonują pojedyncze przeglądanie danych wejściowych od lewej do prawej, podglądając w przód po jednym symbolu terminalnym i konstruując na bieżąco elementy drzewa wyprowadzania jeden po drugim.

Większość metod analizy składniowej należy do jednej z dwóch ogólnych klas, nazywanych metodami zstępującymi (*top-down*) oraz wstępującymi (*bottom-up*). Terminy te odnoszą się do kolejności, w jakiej konstruowane są węzły drzewa rozbioru. W analizatorach zstępujących konstrukcja zaczyna się od korzenia i postępuje w kierunku liści, podczas gdy analizatory wstępujące rozpoczynają od liści i postępują w stronę korzenia. Popularność analizatorów zstępujących wynika z faktu, że łatwiejsze jest konstruowanie wydajnych parserów przy użyciu metod zstępujących. Jednak analiza wstępująca pozwala obsłużyć obszerniejszą klasę gramatyk i schematów translacji, zatem narzędzia programowe do generowania parserów bezpośrednio z gramatyk częściej wykorzystują metody wstępujące.

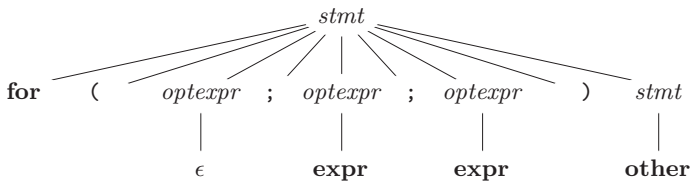
2.4.1. Analiza zstępująca

Pojęcie analizy zstępującej przedstawimy, rozważając gramatykę, która jest dobrze dostosowana do tej klasy metod postępowania. Dalej w tym podrozdziale przeanalizujemy ogólną konstrukcję parserów zstępujących. Gramatyka pokazana na rysunku 2.16 generuje podzbiór instrukcji języka C lub Java. Używamy tu

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \textit{ stmt} \\
 & | & \textbf{for (} \textit{ optexpr ; optexpr ; optexpr } \textbf{)} \textit{ stmt} \\
 & | & \textbf{other} \\
 & & \\
\textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

RYSUNEK 2.16: Gramatyka dla wybranych instrukcji języka C i Javy

wytłuszczonych symboli terminalnych **if** oraz **for** dla słów kluczowych "`if`" i "`for`" odpowiednio, aby podkreślić, że te sekwencje znaków traktowane są jako jednostki, czyli jako pojedyncze symbole terminalne. Dodatkowo terminal **expr** reprezentuje wyrażenia. Bardziej wyczerpująca gramatyka używałaby nieterminalnego symbolu *expr* i zawierała produkcje dla tego nieterminala. Analogicznie, **other** jest terminalem reprezentującym instrukcje o innych konstrukcjach.



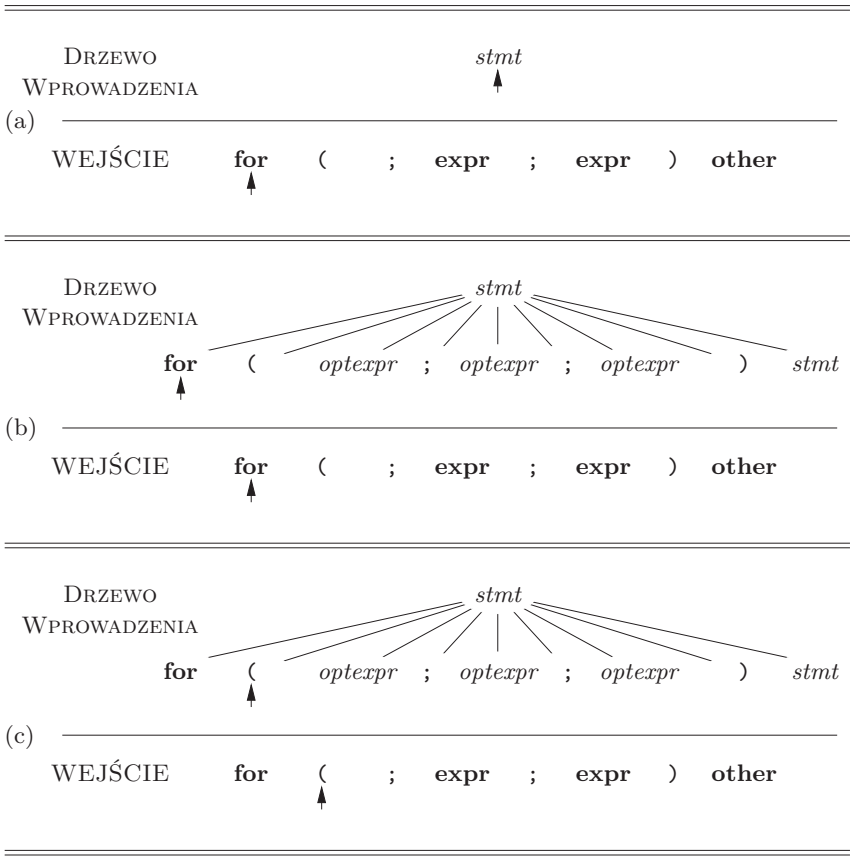
RYSUNEK 2.17: Drzewo rozbioru odpowiadające gramatyce z rysunku 2.16

Konstrukcja zstępująca drzewa rozbioru, podobnego do pokazanego na rysunku 2.17, jest wykonywana przez rozpoczęcie od korzenia opatrzonego etykietą startowego symbolu nieterminalnego *stmt*, a następnie powtarzane jest wykonywanie następujących kroków.

1. W węźle N opatrzonym etykietą nieterminala A wybierz jedną z produkcji dla A i skonstruuj dzieci węzła N dla symboli zawartych w ciele produkcji.
2. Znajdź następny węzeł, w którym poddrzewo musi być konstruowane; typowo pierwszy od lewej jeszcze nierozwinięty nieterminal w drzewie.

Dla niektórych gramatyk powyższe kroki mogą zostać zaimplementowane w czasie pojedynczego przeglądania wejściowego ciągu od lewej do prawej. Bieżący symbol terminalny przeglądany w danych wejściowych często określany jest mianem symbolu podglądanego (*lookahead*). Początkowo symbolem podglądanym jest pierwszy, czyli najbardziej lewy terminal w ciągu wejściowym. Rysunek 2.18 ilustruje konstruowanie drzewa rozbioru z rysunku 2.17 dla ciągu wejściowego.

Terminal **for** jest pierwszym odnalezionym symbolem podglądanym i znana część drzewa składa się z korzenia oznaczonego początkowym nieterminalem *stmt* – rysunek 2.18(a). Celem jest zbudowanie reszty drzewa rozbioru w taki sposób, aby ciąg generowany przez to drzewo pasował do ciągu wejściowego.



RYSUNEK 2.18: Analiza zstępująca podczas skanowania ciągu wejściowego od lewej do prawej

Aby wystąpiło dopasowanie, nieterminalny symbol *stmt* na rysunku 2.18(a) musi generować ciąg, który zaczyna się od symbolu podglądanego **for**. W gramatyce pokazanej na rysunku 2.16 istnieje tylko jedna produkcja dla *stmt*, z której można wyprowadzić taki ciąg, zatem wybieramy właśnie tę i konstruujemy dzieci węzła korzenia oznaczone symbolami z ciała tej produkcji. To rozszerzenie drzewa rozbioru pokazuje rysunek 2.18(b).

Każda z trzech migawek pokazanych na rysunku 2.18 zawiera strzałki oznaczające symbol podglądany w ciągu wejściowym i aktualnie rozważany węzeł w drzewie rozbioru. Po skonstruowaniu dzieci dla węzła kolejnym krokiem jest rozpatrzenie najbardziej lewego dziecka. Na rysunku 2.18(b) zostały właśnie skonstruowane dzieci dla węzła korzenia i analizowany jest skrajny lewy węzeł poziomu dzieci opatrzony etykietą **for**.

Gdy aktualnie analizowany węzeł dotyczy symbolu terminalnego i symbol ten pasuje do symbolu podglądanego z ciągu wejściowego, przechodzimy dalej

zarówno w drzewie rozbioru, jak i w danych wejściowych. Kolejny symbol terminalny z danych wejściowych staje się nowym symbolem podglądanym i rozważamy kolejne dziecko w drzewie. Na rysunku 2.18(c) strzałka została przeniesiona do kolejnego dziecka korzenia, strzałka zaś w danych wejściowych do kolejnego terminala, którym jest **(**. Kolejny krok przeniesie strzałkę w drzewie do dziecka z etykietą nieterminalnego symbolu *optexpr*, a w danych wejściowych do terminala **;**.

W nieterminalnym węźle oznaczonym *optexpr* powtarzamy proces wybierania produkcji dla nieterminala. Produkcje z ϵ jako ciałem („ϵ-produkcje") wymagają specjalnego potraktowania. Na tę chwilę użyjemy ich jako domyślnych, gdy żadna inna produkcja nie może zostać użyta. Powrócimy do tego zagadnienia w podrozdziale 2.4.3. W przypadku nieterminalnego węzła *optexpr* i symbolu podglądanego **;** zostanie użyta ϵ-produkcja, gdyż **;** nie pasuje do jedynej innej produkcji dla *optexpr*, zawierającej jako ciało terminal **expr**.

W ogólności, wybieranie produkcji dla nieterminala może wymagać techniki prób i błędów. Inaczej mówiąc, możemy próbować zastosować pewną produkcję, po czym wycofać się i wypróbować inną, jeśli pierwsza okaże się nieodpowiednia. Produkcja jest nieodpowiednia, jeśli przy jej użyciu nie możemy dokończyć tworzenia drzewa tak, aby pasowało do ciągu wejściowego. Wycofywanie się nie jest jednak potrzebne w ważnym, szczególnym przypadku nazywanym analizą predykcyjną, którą omówimy w kolejnym podrozdziale.

2.4.2. Analiza predykcyjna

Metoda zejść rekurencyjnych to zstępująca metoda analizy składniowej, w której do przetworzenia danych wejściowych wykorzystywany jest zbiór procedur rekurencyjnych. Każda z tych procedur jest powiązana z jednym z nieterminalnych symboli gramatyki. Tutaj rozważamy prostą formę analizy zstępującej opartej na zejściach rekurencyjnych, nazywaną *analizą predykcyjną*, w której symbol podglądany jednoznacznie determinuje przepływ sterowania w ciele procedury dla każdego symbolu nieterminalnego. Sekwencja wywołań procedur w trakcie analizowania ciągu wejściowego jest niejawnie wyznaczona przez drzewo rozbioru dla tego ciągu i może zostać użyta do jawnego zbudowania tego drzewa, jeśli jest to potrzebne.

Parser predykcyjny pokazany na rysunku 2.19 składa się z procedur dla nieterminalnych symboli *stmt* i *optexpr* gramatyki z rysunku 2.16 oraz z dodatkowej procedury dopasowującej *match*, użytej w celu uproszczenia kodu dla *stmt* i *optexpr*. Procedura *match*(t) porównuje swój argument t z symbolem podglądanym i przechodzi do podglądania następnego wejściowego symbolu terminalnego, jeśli stwierdzi dopasowanie. Następnie *match* zmienia wartość zmiennej *lookahead*, zmiennej globalnej, która przechowuje aktualnie podglądany symbol terminalny z ciągu wejściowego.

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

RYSUNEK 2.19: Pseudokod parsera predykcyjnego

Analiza rozpoczyna się od wywołania procedury dla początkowego nieterminalnego symbolu *stmt*. Przy takim samym wejściu, jak na rysunku 2.18, *lookahead* to początkowo pierwszy terminal – **for**. Procedura *stmt* wykonuje kod odpowiadający produkcji

$$stmt \rightarrow \textbf{for}\ (\ optexpr\ ;\ optexpr\ ;\ optexpr\ )\ stmt$$

W kodzie dla ciała produkcji – czyli dla „case **for**" w procedurze *stmt* (czyli dla przypadku odpowiadającego produkcji: $stmt \rightarrow$ **for** (*optexpr*; *optexpr*; *optexpr*) *stmt*) – każdy terminal jest porównywany z symbolem podglądanym, a każdy nieterminal prowadzi do wywołania swojej własnej procedury w następującej sekwencji wywołań:

```
match(for); match('(');
optexpr(); match(';'); optexpr(); match(';'); optexpr();
match(')'); stmt();
```

Predykcyjna analiza opiera się na informacjach o pierwszych symbolach, które mogą być wygenerowane przez ciało produkcji. Mówiąc ściślej, niech α będzie ciągiem symboli gramatycznych (terminali i/lub nieterminali). Definiujemy

FIRST(α) jako zbiór terminali, które występują jako pierwsze symbole w jednym lub więcej ciągów symboli terminalnych wygenerowanych z α. Jeśli α jest równe ϵ lub może wygenerować ϵ, wówczas ϵ również należy do FIRST(α).

Szczegóły, jak wyliczać FIRST(α), zawarte są w podrozdziale 4.4.2. Tutaj po prostu użyjemy rozumowania ad hoc w celu wydedukowania symboli wchodzących w skład FIRST(α). Typowo α albo zaczyna się terminalem, który tym samym jest jedynym elementem FIRST(α), albo α zaczyna się symbolem nieterminalnym, którego ciała produkcji rozpoczynają się terminalami. W tym drugim przypadku te terminale są jedynymi elementami FIRST(α).

Na przykład w odniesieniu do gramatyki z rysunku 2.16, poniżej przedstawiono poprawnie wyznaczone zbiory FIRST

$$\begin{aligned} \text{FIRST}(\textit{stmt}) &= \{\textbf{expr}, \textbf{if}, \textbf{for}, \textbf{other}\} \\ \text{FIRST}(\textbf{expr ;}) &= \{\textbf{expr}\} \end{aligned}$$

Zbiory FIRST muszą być rozpatrywane, jeśli mamy dwie produkcje $A \to \alpha$ oraz $A \to \beta$. Jeśli chwilowo zignorujemy ϵ-produkcje, analiza predykcyjna wymaga, aby zbiory FIRST(α) oraz FIRST(β) były rozłączne. Można wówczas użyć symbolu podglądanego do rozstrzygnięcia, której produkcji użyć. Jeśli symbol należy do FIRST(α), wówczas użyjemy produkcji α. W przeciwnym razie, jeśli symbol podglądany należy do FIRST(β), użyjemy produkcji β.

2.4.3. Kiedy używać ϵ-produkcji

Nasz predykcyjny parser używa ϵ-produkcji jako opcji domyślnej, jeśli żadna inna produkcja nie może zostać użyta. Przy ciągu wejściowym z rysunku 2.18, po dopasowaniu terminali **for** i **(** kolejnym symbolem podglądanym jest **;**. W tym miejscu wywoływana jest procedura *optexpr* i wykonywany jest kod z jej ciała

$$\text{if (} \textit{lookahead} == \textbf{expr} \text{) } \textit{match}(\textbf{expr});$$

Symbol nieterminalny *optexpr* ma dwie produkcje z ciałami **expr** oraz ϵ. Symbol podglądany **;** nie pasuje do terminala **expr**, zatem nie może zostać zastosowana produkcja z ciałem **expr**. W istocie procedura kończy działanie, nie zmieniając symbolu podglądanego i nie robiąc nic innego. Przypadek nierobienia niczego odpowiada zastosowaniu ϵ-produkcji.

Bardziej ogólnie, możemy rozważyć wariant produkcji pokazanych na rysunku 2.16, w której *optexpr* generuje nieterminal odpowiadający wyrażeniu zamiast symbolu terminalnego **expr**:

$$\begin{aligned} \textit{optexpr} &\to \textit{expr} \\ &\mid \epsilon \end{aligned}$$

W tym przypadku *optexpr* albo generuje wyrażenie odpowiadające nieterminalnemu symbolowi *expr*, albo ciąg pusty ϵ. Podczas analizowania *optexpr*, jeśli symbol podglądany nie należy do zbioru FIRST(*expr*), zostanie użyta ϵ-produkcja.

Więcej informacji na temat tego, kiedy należy użyć ϵ-produkcji, zawiera omówienie gramatyk LL(1) w podrozdziale 4.4.3.

2.4.4. Projektowanie parsera predykcyjnego

Spróbujmy teraz uogólnić technikę wprowadzoną nieformalnie w podrozdziale 2.4.2, aby zastosować ją do dowolnej gramatyki, która ma rozłączne zbiory FIRST dla ciał produkcji należących do każdego dowolnego symbolu nieterminalnego. Zobaczymy również, że jeśli mamy schemat translacji – czyli gramatykę z osadzonymi akcjami – możliwe jest wykonanie tych działań jako części procedur zaprojektowanych dla parsera.

Przypomnijmy, że predykcyjny parser to program składający się z procedur dla każdego symbolu nieterminalnego. Procedura dla nieterminala A wykonuje dwie czynności.

1. Decyduje, której *produkcji o symbolu A w nagłówku* należy użyć, badając symbol podglądany. Używana jest produkcja z ciałem α (gdzie α nie jest ϵ – pustym ciągiem), jeśli symbol podglądany należy do zbioru FIRST(α). Jeśli występuje konflikt między dwoma niepustymi ciałami dla dowolnego symbolu podglądanego, nie możemy używać tej metody analizy dla danej gramatyki. Dodatkowo ϵ-produkcja dla A, jeśli istnieje, jest używana, gdy symbol podglądany nie występuje w zbiorze FIRST dla ciała każdej z pozostałych produkcji o symbolu A *w nagłówku*.
2. Następnie procedura odwzorowuje ciało wybranej produkcji. Inaczej mówiąc, ciało produkcji jest „wykonywane" kolejno od lewej. „Wykonanie" symbolu nieterminalnego oznacza wywołanie procedury dla tego nieterminala, terminal zaś pasujący do symbolu podglądanego jest „wykonywany" przez odczytanie kolejnego symbolu wejściowego. Jeśli w jakimś momencie symbol terminalny w ciele nie pasuje do symbolu podglądanego, zgłaszany jest błąd składni.

Rysunek 2.19 pokazuje wynik zastosowania tych reguł do gramatyki z rysunku 2.16.

Podobnie jak schemat translacji tworzony jest przez rozszerzenie gramatyki, translator sterowany składnią można zbudować, rozszerzając parser predykcyjny. Algorytm dla tego zadania zostanie przedstawiony w podrozdziale 5.4. Na razie wystarczy nam następująca ograniczona konstrukcja:

1. Konstruujemy parser predykcyjny, ignorując akcje zawarte w produkcjach.
2. Kopiujemy akcje ze schematu translacji do kodu parsera. Jeśli akcja występuje po symbolu gramatycznym X w produkcji p, jest kopiowana po kodzie implementującym X w procedurze dla p. W przeciwnym razie, jeśli występuje na początku produkcji, jest kopiowana bezpośrednio przed kodem dla ciała tej produkcji.

Translator taki zbudujemy w podrozdziale 2.5.

2.4.5. Lewostronna rekurencja

Istnieje możliwość, aby parser działający metodą zejść rekurencyjnych wpadł w nieskończoną pętlę. Problem pojawia się przy „lewostronnie rekurencyjnych" produkcjach, takich jak

$$expr \quad \rightarrow \quad expr + term$$

gdzie skrajny lewy symbol w ciele produkcji jest taki sam jak nieterminal w jej nagłówku. Załóżmy, że procedura dla *expr* zdecyduje o zastosowaniu tej produkcji. Ciało zaczyna się od *expr*, zatem procedura dla *expr* jest wywoływana rekurencyjnie. Ponieważ symbol podglądany zmienia się tylko wtedy, gdy dopasowany zostanie terminal w ciele produkcji, nie następuje żadna zmiana w danych wejściowych między rekurencyjnymi wywołaniami *expr*. W rezultacie drugie wywołanie *expr* wykonuje dokładnie to samo co pierwsze, a więc po raz trzeci wywołuje *expr* i tak dalej w nieskończoność.

Produkcję lewostronnie rekurencyjną można wyeliminować, przepisując kłopotliwą produkcję. Rozważmy nieterminalny symbol A z dwiema produkcjami

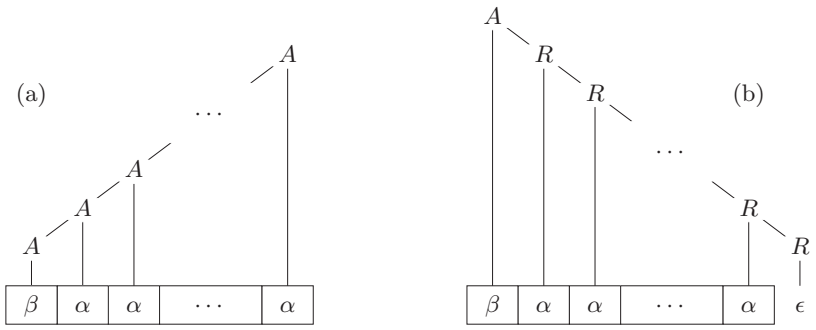
$$A \rightarrow A\alpha \mid \beta$$

gdzie α i β są sekwencjami symboli terminalnych i nieterminalnych, które nie zaczynają się od A. Na przykład w

$$expr \quad \rightarrow \quad expr + term \quad | \quad term$$

nieterminal $A = expr$ ciąg $\alpha = + term$, a ciąg $\beta = term$.

Nieterminalny symbol A i jego produkcję $A \rightarrow A\alpha$ nazywamy lewostronnie rekurencyjnymi, gdyż produkcja $A \rightarrow A\alpha$ zawiera ten sam symbol A, *który jest nagłówkiem procedury,* jako skrajny lewy symbol po prawej stronie[4]. Powtarzane stosowanie tej produkcji buduje sekwencję ciągów α na prawo od A, jak na rysunku 2.20(a). Gdy A zostanie ostatecznie zastąpione przez β, otrzymamyβ, po którym następuje sekwencja zero lub więcej wystąpień α.



RYSUNEK 2.20: Lewo- i prawostronnie rekurencyjne sposoby generowania ciągu

[4] W ogólności, w lewostronnie rekurencyjnej gramatyce, zamiast jawnej produkcji $A \rightarrow A\alpha$, z nieterminalnego symbolu A można wyprowadzić $A\alpha$ przez pośrednie produkcje.

Ten sam efekt można uzyskać, jak pokazuje rysunek 2.20(b), przepisując produkcje dla A w następujący sposób, używając nowego symbolu nieterminalnego R:

$$\begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \alpha R \mid \epsilon \end{array}$$

Nieterminalny symbol R i jego produkcja $R \rightarrow \alpha R$ są *rekurencyjne prawostronnie*, gdyż ta produkcja dla R zawiera symbol R jako ostatni symbol po prawej stronie. Produkcje prawostronnie rekurencyjne prowadzą do drzew, które rozrastają się w dół w prawo, jak na rysunku 2.20(b). Drzewa rozrastające się w prawo sprawiają, że trudniejsze jest tłumaczenie wyrażeń zawierających operatory lewostronnie łączne, takie jak minus. Jednak w podrozdziale 2.5.2 zobaczymy, że nadal można uzyskać poprawne tłumaczenie wyrażeń do notacji postfiksowej dzięki starannemu zaprojektowaniu schematu translacji.

W podrozdziale 4.3.3 rozważymy bardziej ogólne formy rekurencji lewostronnej i pokażemy, jak można wyeliminować ją z gramatyki.

2.4.6. Ćwiczenia do podrozdziału 2.4

Ćwiczenie 2.4.1: Skonstruuj parsery zstępujące oparte na zejściach rekurencyjnych dla poniższych gramatyk:

(a) $S \rightarrow \mathtt{+}\ S\ S \mid \mathtt{-}\ S\ S \mid \mathbf{a}$

(b) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$

(c) $S \rightarrow 0\ S\ 1 \mid 0\ 1$

2.5. Translator dla prostych wyrażeń

Używając technik przedstawionych w trzech ostatnich podrozdziałach, możemy teraz zbudować sterowany składnią translator tłumaczący wyrażenia arytmetyczne na formę postfiksową, jako działający program w Javie. Aby zachować rozsądnie małe rozmiary tego początkowego programu, zaczniemy od wyrażeń zawierających tylko cyfry rozdzielane dwuargumentowymi znakami plus i minus. Później w podrozdziale 2.6 rozszerzymy ten program o tłumaczenie wyrażeń, które zawierają liczby wielocyfrowe i inne operatory. Warto szczegółowo przestudiować tłumaczenie takich wyrażeń, gdyż pojawiają się one jako konstrukcje bardzo wielu języków programowania.

Schemat translacji sterowany składnią często służy jako specyfikacja dla translatora. Schemat pokazany na rysunku 2.21 (powtórzony z rysunku 2.15) definiuje tłumaczenie, które będziemy chcieli wykonać.

Często gramatyka, na której oparty jest dany schemat translacji, musi zostać zmodyfikowana, zanim będzie można ją użyć do analizy przy wykorzystaniu predykcyjnego parsera. W szczególności, gramatyka odpowiadająca schematowi z rysunku 2.21 jest lewostronnie rekurencyjna, a jak widzieliśmy w poprzednim podrozdziale, predykcyjny parser nie może poprawnie obsłużyć takiej gramatyki.

$$
\begin{array}{lcll}
\mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{+}\ \mathit{term} & \{\ \mathrm{print}('\texttt{+}')\ \} \\
 & | & \mathit{expr}\ \texttt{-}\ \mathit{term} & \{\ \mathrm{print}('\texttt{-}')\ \} \\
 & | & \mathit{term} & \\
 & & & \\
\mathit{term} & \rightarrow & \texttt{0} & \{\ \mathrm{print}('\texttt{0}')\ \} \\
 & | & \texttt{1} & \{\ \mathrm{print}('\texttt{1}')\ \} \\
 & & \ldots & \\
 & | & \texttt{9} & \{\ \mathrm{print}('\texttt{9}')\ \}
\end{array}
$$

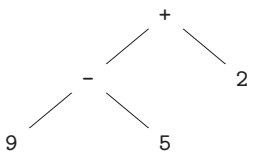
RYSUNEK 2.21: Akcje dla tłumaczenia na notację postfiksową

Wygląda na to, że mamy konflikt: z jednej strony potrzebujemy gramatyki, która umożliwi tłumaczenie; z drugiej potrzebujemy zasadniczo odmiennej gramatyki, która umożliwi analizę składniową. Rozwiązaniem jest rozpoczęcie od gramatyki dla łatwego tłumaczenia i staranne przekształcenie jej w taką, która umożliwi i ułatwi analizę. Dzięki wyeliminowaniu rekurencji lewostronnej z rysunku 2.21 możemy uzyskać gramatykę nadającą się do użycia w predykcyjnym translatorze opartym na zejściach rekurencyjnych.

2.5.1. Składnia abstrakcyjna i konkretna

Użytecznym punktem startowym przy projektowaniu translatora jest struktura danych nazywana *abstrakcyjnym drzewem składniowym* (albo po prostu drzewem składniowym). W takim drzewie dla pewnego wyrażenia każdy wewnętrzny węzeł reprezentuje operator; dzieci tego węzła reprezentują operandy tego operatora. Mówiąc ogólniej, dowolna konstrukcja programowa może zostać obsłużona przez utworzenie operatora dla tej konstrukcji i traktowanie istotnych semantycznie komponentów tej konstrukcji jako operandów.

W abstrakcyjnym drzewie składniowym dla wyrażenia `9-5+2` pokazanym na rysunku 2.22 korzeń reprezentuje operator +. Poddrzewa korzenia odpowiadają podwyrażeniom `9-5` oraz `2`. Zgrupowanie `9-5` jako operandu odzwierciedla przetwarzanie operatorów o tym samym priorytecie od lewej do prawej. Ponieważ – i + mają ten sam priorytet, `9-5+2` jest równoważne `(9-5)+2`.



RYSUNEK 2.22: Drzewo składniowe dla `9-5+2`

Abstrakcyjne drzewa składniowe (lub po prostu drzewa składniowe) do pewnego stopnia przypominają drzewa rozbioru. Jednak w drzewach składniowych wewnętrzne węzły reprezentują konstrukcje programowe, podczas gdy w drzewie rozbioru węzły te odpowiadają symbolom nieterminalnym. Wiele nieterminali gramatyki w istocie odpowiada konstrukcjom programowym, ale inne są tylko

„elementami pomocniczymi" odzwierciedlającymi inną kwalifikację tej samej konstrukcji, na przykład te reprezentujące składniki, czynniki lub inne kwalifikacje tych samych fragmentów wyrażeń. W drzewie składniowym te elementy pomocnicze generalnie nie są potrzebne, a tym samym są pominięte. Aby podkreślić kontrast, drzewo rozbioru niekiedy nazywane jest *konkretnym drzewem składniowym*, gramatyka zaś, na której to drzewo jest oparte – *konkretną składnią* języka.

W drzewie składniowym widocznym na rysunku 2.22 każdy węzeł wewnętrzny jest powiązany z operatorem bez „pomocniczych" węzłów dla tak zwanych produkcji jednostkowych (takich produkcji, których ciało składa się z pojedynczego nieterminala i niczego więcej), jak $expr \rightarrow term$ lub dla ϵ-produkcji, jak $rest \rightarrow \epsilon$.

Pożądane jest, aby schemat translacji bazował na gramatyce, której drzewo rozbioru jest tak podobne do drzewa składniowego, jak to możliwe. Grupowanie podwyrażeń według gramatyki z rysunku 2.21 jest analogiczne do ich grupowania w drzewach składniowych. Na przykład podwyrażenia dla operatora dodawania są formułowane przez *expr* i *term* w ciele produkcji *expr* + *term*.

2.5.2. Dostosowywanie schematu translacji

Technika eliminowania rekurencji lewostronnej naszkicowana na rysunku 2.20 może zostać również zastosowana do produkcji zawierających akcje semantyczne. Technika ta wymaga najpierw rozszerzenia na większą liczbę produkcji dla A. W naszym przykładzie A to *expr* i mamy dla niego dwie produkcje lewostronnie rekurencyjne i jedną, która taka nie jest. Omawiana technika przekształca produkcje $A \rightarrow A\alpha \mid A\beta \mid \gamma$ na

$$\begin{array}{lcl} A & \rightarrow & \gamma R \\ R & \rightarrow & \alpha R \mid \beta R \mid \epsilon \end{array}$$

Następnie musimy przekształcić produkcje, które zawierają osadzone akcje, a nie tylko terminale i nieterminale. Akcje semantyczne osadzone w produkcjach są po prostu przenoszone do przetransformowanych produkcji, tak jakby były terminalami.

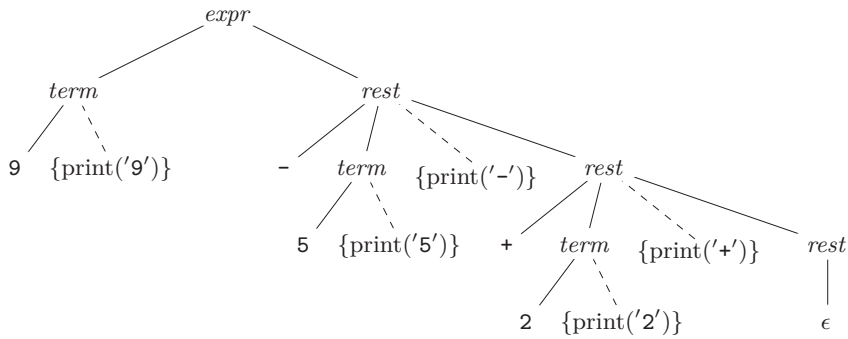
Przykład 2.13: Rozważmy schemat translacji pokazany na rysunku 2.21. Niech

$$\begin{array}{lcl} A & = & expr \\ \alpha & = & \texttt{+}\ term\ \{\ \text{print}(\texttt{'+'})\ \} \\ \beta & = & \texttt{-}\ term\ \{\ \text{print}(\texttt{'-'})\ \} \\ \gamma & = & term \end{array}$$

Następnie transformacja eliminująca rekurencję lewostronną tworzy schemat translacji pokazany na rysunku 2.23. Produkcje dla *expr* z rysunku 2.21 zostały przekształcone na jedną nową produkcję dla *expr*, a nowy nieterminal *rest* odgrywa rolę R. Produkcje dla *term* zostały powtórzone z rysunku 2.21. Rysunek 2.24 pokazuje, jak tłumaczone będzie wyrażenie `9-5+2` przy użyciu gramatyki z rysunku 2.23. ■

$$
\begin{array}{lcl}
expr & \rightarrow & term\ rest \\
rest & \rightarrow & \texttt{+}\ term\ \{\ \text{print}(\texttt{'+'})\ \}\ rest \\
 & | & \texttt{-}\ term\ \{\ \text{print}(\texttt{'-'})\ \}\ rest \\
 & | & \epsilon \\
term & \rightarrow & \texttt{0}\ \{\ \text{print}(\texttt{'0'})\ \} \\
 & | & \texttt{1}\ \{\ \text{print}(\texttt{'1'})\ \} \\
 & & \ldots \\
 & | & \texttt{9}\ \{\ \text{print}(\texttt{'9'})\ \}
\end{array}
$$

RYSUNEK 2.23: Schemat translacji po wyeliminowaniu rekurencji lewostronnej



RYSUNEK 2.24: Tłumaczenie wyrażenia `9-5+2` na `95-2+`

Eliminowanie rekurencji lewostronnej musi być wykonywane ostrożnie, aby zagwarantować, że zachowamy właściwą kolejność akcji semantycznych. Na przykład przekształcony schemat z rysunku 2.23 zawiera akcje { print('+') } oraz { print('-') } w środku produkcji w każdym przypadku, między nieterminalnymi symbolami *term* i *rest*. Gdyby akcje te zostały przeniesione na koniec, po *rest*, wówczas tłumaczenie okazałoby się nieprawidłowe. Pozostawiamy czytelnikowi wykazanie, że `9-5+2` zostałoby wówczas przetłumaczone nieprawidłowo na `952+-` – notację postfiksową dla `9-(5+2)`, a nie na pożądane `95-2+`, czyli prawidłową notację postfiksową dla `(9-5)+2`.

2.5.3. Procedury dla nieterminali

Funkcje *expr*, *rest* oraz *term* na rysunku 2.25 implementują sterowany składniowo schemat translacji z rysunku 2.23. Funkcje te naśladują ciała produkcji dla odpowiadających im nieterminali. Funkcja *expr* implementuje produkcję $expr \rightarrow term\ rest$ przez wywołanie *term*(), po czym następuje wywołanie *rest*().

Funkcja *rest* implementuje trzy produkcje dla nieterminala *rest* z rysunku 2.23. Stosuje pierwszą produkcję, jeśli symbol podglądany (*lookahead*) jest znakiem plus, drugą produkcję, gdy jest to znak minus, oraz produkcję $rest \rightarrow \epsilon$ we wszystkich pozostałych przypadkach. Pierwsze dwie produkcje dla *rest* są

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* nic nie rób z ciągiem wejściowym */;
}

void term() {
        if ( lookahead is a digit ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("błąd składniowy"');
}
```

RYSUNEK 2.25: Pseudokod dla nieterminali *expr*, *rest* i *term*.

implementowane w dwóch pierwszych rozgałęzieniach wyrażenia if w procedurze *rest*. Jeśli symbol podglądany to +, znak plus jest dopasowywany przez wywołanie *match*('+'). Po wywołaniu *term*() wykonywana jest akcja semantyczna przez wypisanie znaku plus. Druga produkcja jest analogiczna, przy czym zastępujemy znak + przez znak - (minus). Ponieważ trzecia produkcja dla *rest* zawiera ϵ jako prawą stronę, ostatnia klauzula else funkcji *rest* nie robi nic.

Dziesięć produkcji dla *term* generuje dziesięć cyfr. Ponieważ każda z tych produkcji generuje cyfrę i ją wypisuje, ten sam kod z rysunku 2.25 implementuje je wszystkie. Jeśli test się powiedzie, zmienna t przechowuje cyfrę reprezentowaną przez *lookahead*, dzięki czemu może zostać wypisana po wywołaniu *match*. Zauważmy, że *match* zmienia symbol podglądany w zmiennej *lookahead*, zatem cyfra musi być zachowana, aby mogła zostać później wypisana[5].

2.5.4. Upraszczanie translatora

Zanim pokażemy pełny program, powinniśmy wykonać dwa upraszczające przekształcenia kodu z rysunku 2.25. Uproszczenia te wstawią procedurę *rest* do procedury *expr*. Gdy tłumaczone są wyrażenia o wielu poziomach pierwszeństwa wykonywania działań, tego typu uproszczenia pozwalają zredukować liczbę potrzebnych procedur.

[5] Jako pomniejszą optymalizację moglibyśmy wykonać print przed wywołaniem *match*, aby uniknąć konieczności zachowania cyfry. Jednak w ogólności zmienianie kolejności działań i symboli gramatycznych jest ryzykowne, gdyż może zmienić to, co ma robić tłumaczenie.

Po pierwsze, określone wywołania rekurencyjne mogą zostać zastąpione iteracjami. Gdy ostatnia instrukcja wykonywana w ciele procedury jest rekurencyjnym wywołaniem tej samej procedury, to wywołanie takie nazywamy ogonowo rekurencyjnym (*tail recursive*). Na przykład w funkcji *rest* wywołania *rest*() dla symbolu podglądanego + i - są wywołaniami ogonowo rekurencyjnymi, gdyż w każdej z tych gałęzi kodu wywołanie *rest* jest ostatnią instrukcją wykonywaną w tym wywołaniu *rest*.

W przypadku procedury bez parametrów rekurencja ogonowa może zostać zastąpiona po prostu skokiem do początku procedury. Kod dla *rest* może zostać zatem przepisany tak, jak w pseudokodzie pokazanym na rysunku 2.26. Dopóki symbol podglądany to znak plus lub minus, procedura *rest* dopasowuje znak, wywołuje *term* w celu dopasowania cyfry i kontynuuje proces. W przeciwnym razie wychodzi z pętli while i następuje powrót z procedury *rest*.

```
void rest() {
        while( true ) {
                if( lookahead == '+' ) {
                        match('+'); term(); print('+'); continue;
                }
                else if ( lookahead == '-' ) {
                        match('-'); term(); print('-'); continue;
                }
                break;
        }
}
```

RYSUNEK 2.26: Eliminowanie rekurencji ogonowej w procedurze *rest* z rysunku 2.25

Po drugie, gotowy program w Javie wymaga jeszcze jednej zmiany. Po tym, jak rekurencyjne wywołania *rest* z rysunku 2.25 zostały zastąpione iteracjami, jedyne pozostałe wywołanie procedury *rest* znajduje się w procedurze *expr*. Obie procedury mogą zostać zatem zintegrowane w jedną przez zastąpienie wywołania *rest*() ciałem procedury *rest*.

2.5.5. Kompletny program

Pełny program naszego translatora w języku Java jest widoczny na rysunku 2.27. Pierwszy wiersz tego kodu, zaczynający się od `import`, zapewnia dostęp do pakietu `java.io` zawierającego systemowe procedury wejścia i wyjścia. Reszta kodu składa się z dwóch klas `Parser` oraz `Postfix`. Klasa `Parser` zawiera zmienną `lookahead` oraz funkcje `Parser`, `expr`, `term` i `match`.

Wykonywanie rozpoczyna się od funkcji `main`, zdefiniowanej w klasie `Postfix`. Funkcja `main` tworzy wystąpienie `parse` klasy `Parser` i wywołuje jego funkcję `expr` w celu analizy wyrażenia.

Funkcja `Parser` o tej samej nazwie co jej klasa jest *konstruktorem*; jest wywoływana automatycznie podczas tworzenia obiektu tej klasy. Można zauważyć w jej definicji na początku klasy `Parser`, że konstruktor `Parser` inicjuje zmienną `lookahead`, odczytując token. Tokeny składające się z pojedynczych znaków są dostarczane przez systemową procedurę wejściową `read`, która wczytuje kolejny znak z pliku wejściowego. Zwróćmy uwagę, że zmienna `lookahead` jest zadeklarowana jako liczba całkowita, a nie znak, przewidując fakt, że w dalszej części wprowadzimy inne tokeny niż pojedyncze znaki.

Funkcja `expr` jest wynikiem uproszczeń omówionych w podrozdziale 2.5.4. Implementuje ona symbole nieterminalne *expr* i *rest* z rysunku 2.23. Kod dla `expr` wywołuje `term`, a następnie zawiera pętlę while, która w nieskończoność sprawdza, czy `lookahead` jest równa `'+'` lub `'-'`. Sterowanie jest wyprowadzane z tej pętli, gdy natrafi na instrukcję powrotu. Wewnątrz pętli funkcje wejścia/wyjścia klasy `System` są używane do wypisania znaku.

Funkcja `term` używa procedury `isDigit` z klasy `Character` w celu przetestowania, czy symbol podglądany jest cyfrą. Procedura `isDigit` oczekuje jednak, że zostanie zastosowana do znaku, podczas gdy `lookahead` została zadeklarowana dla typu całkowitoliczbowego, przewidując przyszłe rozszerzenie. Konstrukcja `(char)lookahead` rzutuje (przekształca) `lookahead` na typ znakowy. Jako niewielka zmiana w stosunku do rysunku 2.25 semantyczna akcja wypisania znaku podglądanego w funkcji `term` następuje przed wywołaniem funkcji `match`.

Funkcja `match` sprawdza symbole terminalne; odczytuje następny terminal z wejścia, jeśli symbol podglądany został dopasowany i sygnalizuje błąd w innym przypadku, wykonując

```
throw new Error("syntax error");
```

Kod ten tworzy nowy wyjątek z klasy `Error` i dostarcza do niego ciąg `syntax error` jako komunikat błędu. Język Java nie wymaga deklarowania wyjątków klasy `Error` w klauzuli `throws`, gdyż zakłada się, że będą one używane tylko w nietypowych zdarzeniach, które nigdy nie powinny wystąpić [6].

[6] Obsługę błędów można usprawnić przy użyciu funkcji obsługi wyjątków w Javie. Jednym z możliwych podejść jest zdefiniowanie nowego wyjątku, na przykład `SyntaxError`, rozszerzającego systemową klasę `Exception`. A zatem rzucamy wyjątek `SyntaxError` zamiast `Error`, gdy błąd zostanie wykryty w funkcji `term` albo `match`. Później możemy obsłużyć ten wyjątek w funkcji `main`, umieszczając wywołanie `parse.expr()` w instrukcji `try`, która przechwytuje wyjątek `SyntaxError`, wypisuje komunikat i kończy działanie programu. Musielibyśmy dodać do programu z rysunku 2.27 klasę `SyntaxError`. Aby dopełnić rozszerzenie, oprócz `IOException`, funkcje `match` i `term` muszą teraz deklarować, że mogą rzucić wyjątek `SyntaxError`. Funkcja `expr`, która je wywołuje, również musi deklarować, że może rzucić wyjątek `SyntaxError`.

```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead =='+' ) {
                match('+'); term(); System.out.write('+');
            }
        else if( lookahead =='-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
        }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
    }
}
```

RYSUNEK 2.27: Program w Javie tłumaczący wyrażenia infiksowe na formę postfiksową

Kilka uderzających cech języka Java

Osoby słabo znające język Java mogą uznać poniższe uwagi za przydatne przy odczytywaniu kodu z rysunku 2.27:

- Klasa w Javie składa się z sekwencji definicji zmiennych i funkcji.
- Nawiasy obramowujące listę parametrów funkcji są niezbędne nawet wtedy, gdy funkcja nie przyjmuje żadnych parametrów; dlatego piszemy `expr()` i `term()`. Funkcje te są w rzeczywistości procedurami, gdyż nie zwracają wartości, co jest sygnalizowane słowem kluczowym `void` przed nazwą funkcji.
- Funkcje komunikują się albo przekazując parametry „przez wartość", albo przez dostęp do współdzielonych danych. Na przykład funkcje `expr()` i `term()` badają symbol podglądany, używając zmiennej `lookahead`, do której mogą mieć dostęp, gdyż wszystkie (zmienna i te funkcje) należą do tej samej klasy `Parser`.
- Podobnie jak C, Java używa znaku `=` dla instrukcji przypisania, `==` dla sprawdzenia równości oraz `!=` dla sprawdzenia nierówności (różności).
- Klauzula `"throws IOException"` w definicji funkcji `term()` deklaruje, że może wystąpić wyjątek o nazwie `IOException`. Taki wyjątek następuje wtedy, gdy nie ma danych wejściowych do odczytania, gdy funkcja `match` używa procedury `read`. Dowolna funkcja, która wywołuje `match`, również musi deklarować, że wyjątek `IOException` może pojawić się podczas jej wykonywania.

2.6. Analiza leksykalna

Analizator leksykalny, czyli *lekser*, odczytuje znaki z wejścia i grupuje je w „obiekty tokenów". Wraz z symbolem terminalnym, który jest używany w decyzjach analizy składniowej, obiekt tokenu niesie dodatkowe informacje w postaci wartości atrybutów. Jak dotąd nie rozróżnialiśmy terminów „token" i „terminal" (symbol terminalny), gdyż parser ignoruje wartości atrybutów zawarte w tokenie. W tym podrozdziale token to symbol terminalny (dla parsera) wraz z dodatkowymi informacjami.

Sekwencja znaków wejściowych składających się na pojedynczy token (rozpoznawana przez lekser jako pojedynczy token) nazywana jest *leksemem*. Możemy więc powiedzieć, że analizator leksykalny izoluje parser od leksemowej reprezentacji tokenów. Lekser przedstawiany w tym podrozdziale pozwala na pojawianie się w wyrażeniach liczb, identyfikatorów oraz „białych znaków", czyli odstępów (spacji, tabulatorów i znaków nowego wiersza). Może zostać wykorzystany do rozbudowy translatora wyrażeń z poprzedniego podrozdziału. Ponieważ konieczne jest rozszerzenie gramatyki wyrażeń z rysunku 2.21, aby pozwolić na stosowanie wielocyfrowych liczb i identyfikatorów, wykorzystamy tę okazję do wprowadzenia operacji mnożenia i dzielenia. Rozszerzony schemat translacji pokazany jest na rysunku 2.28.