POLITECHNIKA CZĘSTOCHOWSKA
WYDZIAŁ INŻYNIERII MECHANICZNEJ I INFORMATYKI
INSTYTUT INFORMATYKI TEORETYCZNEJ I STOSOWANEJ

PRACA MAGISTERSKA

METODY ANALIZY SYKTANTYCZNEJ I SEMANTYCZNEJ ORAZ GENEROWANIE KODU ASEMBLERA DLA ARCHITEKTURY INTEL x86

Syntax and semantics analysis methods
and assembler code generating for Intel x86 architecture

Łukasz Czerwiński
numer albumu: 38825
kierunek: Informatyka

Promotor: dr inż. Grzegorz Szwarc

Częstochowa, 2007

Spis treści

1 Wstęp

Kompilator jest programem, który czyta kod napisany w jednym języku określanym jako język źródłowy i tłumaczy go na równoważny program w drugim języku wynikowym. W przypadku, kiedy translacja ta jest niemożliwa kompilator powinien jak najprecyzyjniej określić przyczynę i powiadomić o tym użytkownika.

Pierwszym kompilatorem był prawdopodobnie Autocoder [27] napisany w 1952. W latach 50 oraz 60 XX wieku, bardzo intensywnie przebiegały badania nad zasadą działania kompilatorów. W 1958 Strong prowadził badania nad podziałem procesu kompilacji na dwie główne części oraz podziału kompilatora na jego „przód" i „tył". Idea ta jest stosowana aż do dnia dzisiejszego. Przód kompilatora jest odpowiedzialny za analizę kodu źródłowego, czyli jego wczytanie, sprawdzenie jego poprawności gramatycznej oraz logicznej. Następnie generowany jest równoważny program w reprezentacji pośredniej. Strong zaproponował w tym celu pojęcie uniwersalnego języka komputerowego UNCOL [2], lecz ten pomysł jest wciąż niedoścignionym ideałem. Z reprezentacji pośredniej, w fazie syntezy wykonywanej przez tył kompilatora, tworzony jest kod specyficzny dla maszyny docelowej.
Dzięki takiemu podziałowi ról, rozgranicza się strukturę języka od cech maszyny, na którą będą pisane w tym języku programy.

Etapy kompilacji, analiza i synteza, chociaż uprościły cały proces translacji ciągle są etapami bardzo złożonymi. Wygodnym podejściem podczas implementowania kompilatorów jest ich dalszy podział na mniejsze i bardziej wyspecjalizowane fragmenty.

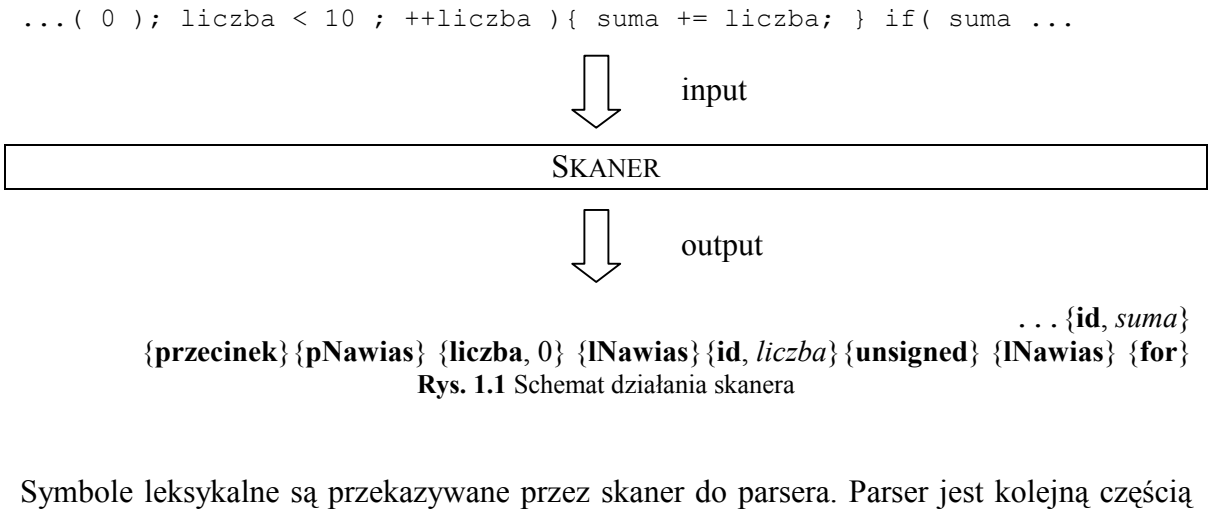
Fazę analizy można podzielić na analizę leksykalną, składniową oraz semantyczną, podczas gdy syntezę na generowanie kodu pośredniego, optymalizację i generowanie kodu docelowego.

Przygotowaniem do procesu kompilacji jest preprocesing. Preprocesing zazwyczaj nie jest wykonywany przez kompilator, lecz przez osobny program – preprocesor. W trakcie tego etapu dokonuje się dołączania plików nagłówkowych. Dzięki temu kompilator dostaje jeden duży plik do kompilacji[1]. Plik ten najczęściej nie zawiera już komentarzy, które również mogą być podczas tej fazy usunięte. W zależności od możliwości preprocesora, można dokonać również rozwijania makrodefinicji a używając kompilacji warunkowej zmieniać kod, który zostanie poddany kompilacji.

Pierwszym etapem kompilacji jest analiza leksykalna (ang. *lexical analysis*). Wykonywana jest przez część kompilatora zwana skanerem (ang. *scanner*). Skaner wczytując kod programu znak po znaku grupuje je w większe jednostki zwane symbolami leksykalnymi. Typowymi symbolami leksykalnymi w językach programowania są **identyfikator**, **liczbaRzeczywista**, **liczbaCałkowita**, **operatorDodawania**. W trakcie tego etapu kompilacji można wykryć pierwsze proste błędy popełnione przez użytkownika. Ponieważ w danej chwili widzi tylko jeden symbol leksykalny, oraz nie posiada pamięci, co do symboli już wczytanych, błędy te odnoszą się do aktualnie wczytywanego symbolu. Przykładem takich błędów może być niepoprawny identyfikator użyty jako nazwa zmiennej. Język C/C++ wymaga, aby identyfikator składał się z litery, po której następuje dowolna liczba liter lub cyfr. W wyniku tych reguł, jeśli w kodzie programu zostanie rozpoznany napis *zmienna$tymczasowa*, a znak dolara $ nie należy do zbioru liter wiadomo, że ten identyfikator jest na pewno niepoprawny. Również liczba 0xFFFFFG jest

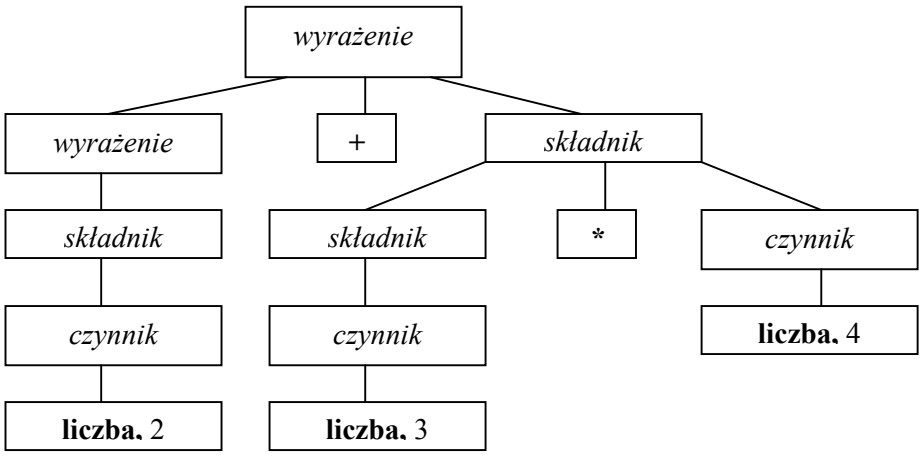
[1] Kompilator g++ umożliwia wykonanie proprocesingu opcją –E (*g++ main.cpp –E > plik.cpp*)

błędna. Zaczyna się od znaków 0x, które sugerują zapis w notacji szesnastkowej, jednak znak G nie jest poprawną cyfrą szesnastkową. Zbiór reguł opisujący wygląd symboli leksykalnych jest na tyle prosty, że wystarcza użycie wyrażeń regularnych, aby je opisać.
Zasada działania skanera została przedstawiona na Rys. 1.1. Kod programu jest czytany i skaner znajduje w nim symbole leksykanle, które są przekazywane go parsera. Niektóre symbole leksykalne mają również swoje wartości jak np.: symbol **id** i jego wartość *suma*.



Rys. 1.1 Schemat działania skanera

Symbole leksykalne są przekazywane przez skaner do parsera. Parser jest kolejną częścią kompilatora i jest odpowiedzialny za analizę składniową (ang. *syntax analysis*). Analiza składniowa sprawdza, czy program jest poprawny z gramatyką. Wyrażenia regularne, choć wystarczają do opisu symboli leksykalnych mają za małą siłę wyrazu, aby mogły być użyte do opisu składni języka. Wyrażenie regularne nie potrafi opisać struktur zagnieżdżonych, takich jak sparowane nawiasy, sparowane znaczniki początku i końca bloków ({, }, begin, end) itd. Gramatyka języków formalnych z natury ma budowę rekurencyjną i poprzez niejawne utrzymywanie stosu doskonale radzi sobie z takim zadaniem. Parser i skaner tworzą parę typu producent – konsument. Parser wysyła żądanie do skanera o kolejny symbol leksykalny programu, a skaner mu go dostarcza. Parser składa symbole leksykalne w drzewo składniowe. Jeżeli na wejściu nie ma już żadnych symboli rozpoznanych przez skaner, a parser zakończył budowanie drzewa składniowego, faza parsowania jest zakończona pomyślnie i można przejść do kolejnego etapu – analizy semantycznej (logicznej). Przykład drzewa składniowego przedstawiono na Rys. 1.2.



Rys. 1.2 Drzewo składniowe dla wyrażenia 2 + 3 * 4

Celem kolejnego etapu kompilacji jest analiza semantyczna kodu użytkownika. Analizator semantyczny dysponuje już kompletnym drzewem składniowym i dlatego o wiele łatwiej niż parser może odnaleźć wiele różnych błędów. Język C/C++ wymaga, żeby każdy symbol zanim zostanie użyty w programie został najpierw zadeklarowany. Gramatyki używane w językach programowania nie są w stanie zapewnić tego warunku. Również błędy jak niezgodność typów, przypisanie na stałą lub błędne instrukcje sterujące muszą zostać wykryte przez fazę analizy semantycznej. Oprócz wyszukiwania błędów logicznych w programie, faza semantyczna może ustalać sposób zachowania programu. Jeżeli język pozwala na przeciążanie funkcji, stosowanie zasięgów i przestrzeni nazw, używanie klas z polimorfizmem ten etap kompilacji jest najodpowiedniejszy, aby określić działanie programu. Faza analizy semantycznej jest ostatnią fazą wykonywaną przez przód kompilatora.
Na Rys. 1.3 przedstawiono fragment większego programu w języku C++. Zadaniem fazy semantycznej jest sprawdzenie, czy program można skompilować, oraz określenie, że wywołana ma zostać funkcja `function /*1*/` a nie metoda `function /*2*/`.

```
void function(); /*1*/

template< typename _T >
struct Base {
      void function();/*2*/
};

template< typename _T >
struct Derived : Base< _T > {
      void other_function(){
            return function();
      }
};
```

Rys. 1.3 Kod w języku C++ z przesłanianiem nazw

Jeżeli program użytkownika został już wczytany i sprawdzony pod kątem wystąpienia przeróżnych błędów, na podstawie drzewa składniowego można dokonać generowania kodu pośredniego a następnie kodu docelowego.

Kod pośredni jest reprezentacją programu w bardzo prostym języku z użyciem krótkiej liczby prostych instrukcji. Popularnym rodzajem kodu pośredniego jest kod trójadresowy, który dopuszcza użycie co najwyżej trzech argumentów w jednej instrukcji. Pożądaną cechą kodu pośredniego jest jego prostota oraz łatwość translacji do kodu wynikowego. Na Rys. 1.4 przedstawiono fragment programu zapisanego w kodzie trójadresowym.

```
temp1 = inttoreal( 50 )
temp2 = id1 * temp1
temp3 = id2 + temp2
id3 = temp3
```

Rys. 1.4 Instrukcje zapisane w kodzie trójadresowym

Ponieważ złożoność programów oraz maszyn, na które te programy są tworzone rośnie w bardzo szybkim tempie, programiści często nie są w stanie zoptymalizować kodu na poziomie pojedynczych instrukcji maszynowych. Dlatego również to zadanie powierza się kompilatorom. Optymalizacja kodu pośredniego jest ważnym i bardzo trudnym etapem kompilacji. Często należy przyjąć kompromis pomiędzy optymalizacją prędkości programu, a jego zapotrzebowaniem na pamięć operacyjną. Na Rys. 1.5 pokazano przykład programu napisanego w C++. Tabela 1 zawiera jego skompilowaną do asemblera wersję bez optymalizacji oraz z optymalizacją. Już w tak krótkim programie można zauważyć, że optymalizacja może zwiększyć prędkość działania programu.

```
int main( int argc, char** argv ) {
        return 0;
}
```

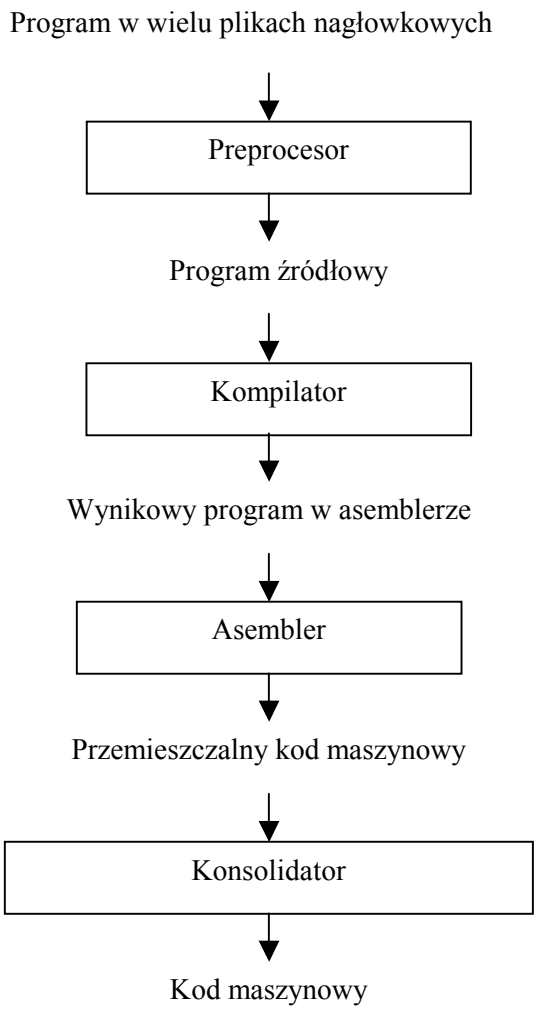
Rys. 1.5 Program w języku C++

Tabela 1 Program z Rys. 1.5 skompilowany bez oraz z optymalizacją

; program skompilowany kompilatorem ; g++-3.4.1 bez optymalizacji (opcja –O0)	; program skompilowany kompilatorem ; g++-3.4.1 z optymalizacją (opcja –O2)

`.file "main.cpp"`
`.def ___main;`
`.scl 2;`
`.type32;`
`.endef`
`.text`
`.align 2`
`.globl _main`
`.def _main;`
`.scl 2;`
`.type 32;`
`.endef`
`_main:`
`pushl %ebp`
`movl %esp, %ebp`
`subl $8, %esp`
`andl $-16, %esp`
`movl $0, %eax`
`addl $15, %eax`
`addl $15, %eax`
`shrl $4, %eax`
`sall $4, %eax`
`movl %eax, -4(%ebp)`
`movl -4(%ebp), %eax`
`call __alloca`
`call ___main`
`movl $0, %eax`
`leave`
`ret`
	`.file "main.cpp"`
	`.def ___main;`
	`.scl 2;`
	`type 32; .endef`
	`.text`
	`.align 2`
	`.p2align 4,,15`
	`.globl _main`
	`.def _main;`
	`.scl 2;`
	`.type 32;`
	`.endef`
	`_main:`
	`pushl %ebp`
	`movl $16, %eax`
	`movl %esp, %ebp`
	`subl $8, %esp`
	`andl $-16, %esp`
	`call __alloca`
	`call ___main`
	`leave`
	`xorl %eax, %eax`
	`ret`

Ostatnim etapem kompilacji jest generowanie kodu maszynowego. Kompilator może wygenerować kod języka asemblera i następnie posłużyć się kolejnym programem z celu zamiany go na kod maszynowy lub stworzyć kod maszynowy samodzielnie. Język asemblera jest językiem mnemonicznym, co oznacza, że pojedyncza instrukcja asemblera jest równoważna pojedynczej instrukcji maszynowej.
Do utworzenia pliku wykonywalnego, oprócz samego kompilatora, może być również potrzebne użycie innych programów.
Program źródłowy może być podzielony na moduły przechowywane w oddzielnych plikach. Łaczeniem tych plików zajmuje się preprocesor.
Kod gotowy do kompilacji przetwarzany jest przez kompilator, który może wygenerować jedynie kod asemblera.
Przemieszczalny kod maszynowy z asemblera tworzy program zwany asemblerem.
Konsolidator (ang. *linker*) umożliwia łączenie wielu plików zawierających przemieszczalny kod maszynowy w pojedynczy program.



Rys. 1.6 System przetwarzania języków

Rysunek Rys. 1.6 przedstawia kontekst procesu kompilacji. Jest to środowisko, które tłumaczy program napisany w języku programowania do kodu maszynowego zdolnego do wykonania przez procesor. Jest to proces długi i skomplikowany, a kompilacja jest tylko jego fragmentem.

2 Cel i zakres pracy

Języki programistyczne można podzielić na wysokiego i niskiego poziomu. Języki niskiego poziomu opisują możliwości maszyny. Są zależne od architektury procesora co czyni je nieprzenośnymi. Języki te sprowadzają się do podstawowych instrukcji procesora, co czyni programy napisane w nich długimi i trudnymi w zrozumieniu.
Języki wysokiego poziomu, dostarczają mechanizmów które upraszczają składnię oraz semantykę programu. Programy pisane w nich są bardziej zwięzłe oraz prostsze do czytania i modyfikowania. Są zrozumiałe dla człowieka, ale nie dla maszyny.
Wygodnym kompromisem pomiędzy pisaniem przenośnego programu zrozumiałego dla człowieka a kodem maszynowym wymaganym przez procesor jest automatyczne przekształcenie języka wysokiego poziomu do kodu maszynowego.

Podstawowym celem pracy jest translacja programu zapisanego w języku wysokiego poziomu na kod asemblera.

Zrealizowanie podstawowego założenia wymaga przyjęcia i zrealizowania celów cząstkowych, którymi są:

- zdefiniowanie języka wewnętrznego
    - zdefiniowanie symboli leksykalnych
    - zdefiniowanie gramatyki
    - zdefiniowanie akcji semantycznych
- określenie architektury docelowej dla generowanego asemblera
- dostarczenie aplikacji tłumaczącej język wewnętrzny na kod asemblera
- dostarczenie środowiska do linkowania wygenerowanego kodu asemblera
- dostarczenie przykładowych programów w języku wewnętrznym.

Na etapie analizy wymagań stawianych aplikacji przyjęto następujące założenia co do funkcjonalności wewnętrznego języka

1. Obsługa wyrażeń matematycznych wykonywanych na liczbach całkowitych
2. Obsługa mechanizmu tworzenia zmiennych lokalnych
3. Obsługa mechanizmu wywoływania funkcji
    a. Do funkcji można przekazać dowolną liczbę parametrów
    b. Funkcja może zwracać opcjonalnie jedną wartość
    c. Funkcje można wywoływać rekurencyjnie
4. Obsługa instrukcji iteracyjnych
    a. Pętla `for`
    b. Pętla `while`
5. Obsługa instrukcji warunkowej `if else`
6. Wykrycie błędów semantycznych, oraz zgłoszenie ich użytkownikowi
    a. Użycie niezdefiniowanego symbolu
    b. Powtórne zdefiniowanie symbolu
    c. Niezgodność typów w wyrażeniach matematycznych

Oprócz wymagań funkcjonalnych postawiono wymagania odnośnie jakości kodu. W tym celu w kodzie użyto następujące mechanizmy i techniki

1. Asercje
2. Inteligentne wskaźniki z biblioteki `boost`
3. Wzorce projektowe, `Factory`, `Singleton` , `Visitor`, `Curious Recurring Template`, `Adaptor`

Aplikacji na etapie projektowania nie stawiano zadania spełnienia wysokich wymagań wydajnościowych zarównoo do prędkości działania samego kompilatora jak i do efektywności generowanego kodu asemblera, zarówno pod kątem wymagań pamięciowych jak i prędkości działania.

Praca składa się ze wstępu, siedmiu rozdziałów oraz spisu literatury.

Rozdział pierwszy poświęcony jest analizie leksykalnej. Zaprezentowano w nim automaty stanowe jako wygodny model analizy napisów.

Rozdział drugi przedstawia podstawowe elementy z teorii języków formalnych. Opisuje języki, gramatyki oraz dwie najbardziej popularne metody parsowania TOP-DOWN i BOTTOM-UP.

W rozdziale trzecim opisano znaczenie akcji semantycznych podczas kompilacji. Skoncentrowano się na pojęciu tablicy symboli, oraz przedstawiono przykładowe mechanizmy stosowane podczas implementacji dziedziczenia w języku C++.

Rozdział czwarty jest wstępem do architektury komputerów. Tematem pracy jest omówienie metod parsowania, analizy semantycznej oraz generowania kodu asemblera dla architektury Intel x86. Jednak żeby nadać pracy bardziej badawczy charakter, do analizy przyjęto trzy procesory: Intel 8051, Intel Pentium IV, UltraSPARC III. Każdemu z tych procesorów stawia się odmienne wymagania, co przekłada się na różnorodne ich architektury. Należy również podkreślić, że celem kompilatora dołączonego do pracy jest generowanie kodu asemblera tylko dla architektury Intel x86, tak jak zostało wspomniane w temacie pracy.

Elementy asemblera przedstawiono w rozdziale piątym. Przedstawiono w nim instrukcje a także sposób translacji wysokopoziomowych instrukcji na kod asemblera oraz przykładowe programy dla architektury UltraSPARC III oraz Intel x86 w notacji AT&T.

Rozdział szósty poświęcony został opisowi przykładowej aplikacji, kompilatorowi `mtc` (*Master Thesis Compiler*). Zaprezentowano w nim sposób użycia kompilatora, język wewnętrzny a także ogólny zarys budowy aplikacji i użytych w nim rozwiązań technik obiektowych.

Praca podsumowana została w rozdziale siódmym. Rozdział zawiera uwagi końcowe oraz spostrzeżenia na temat zastosowanego algorytmu translacji wyrażeń arytmetycznych.

Praca kończy się spisem literatury zawierającym 26 pozycji.

3 Analizy leksykalna

Zadaniem analizy leksykalnej (ang. *lexical analysis*) jest znalezienie w kodzie źródłowym symboli leksykalnych i przekazanie ich do dalszych etapów kompilacji. Z niektórymi symbolami leksykalnymi skojarzona jest *wartość leksykalna*. Przykładowo wartością leksykalną symbolu **identyfikator** może być *zmienna*, *licznik* natomiast symbol **liczbaRzeczywista** może mieć wartości takie jak *2*, *-2e2*, *3.1415*. Analizator leksykalny (ang. *scaner*) może wykonywać także inne zadania. Do najbardziej popularnych należy usuwanie białych znaków z kodu źródłowego i komentarzy. Również *preprocessing* z kompilacją warunkową i załączaniem niezbędnych plików może zostać wykonany w tej fazie kompilacji.

3.1 Napisy i języki

Symbol jest obiektem abstrakcyjnym którego, podobnie jak punkt lub prosta w geometrii, nie definiuje się w sposób formalny. Często używanymi symbolami są na przykład litery lub cyfry. *Alfabet* (ang. *alphabet*) lub *słownik* (ang. *dictionary*) jest niepustym, skończonym zbiorem symboli. Formalna definicja alfabetu

$$T \text{ będzie alfabetem} \Leftrightarrow T \neq \emptyset, \#T < \infty$$

Typowymi alfabetami są alfabet języka polskiego {a, ą, b, c, ć, d, e, ę, f, g, h, i, j, k, l, ł, m, n, ń, o, ó, p, r, s, ś, t, u, w, y, z, ż, ź} lub alfabet binarny {0, 1}. Przykładem alfabetu komputerowego może być zbiór znaków ASCII oraz EBCDIC.

Napis (ang. *string*) nad pewnym alfabetem jest skończonym ciągiem symboli należących do danego alfabetu. W teorii języków synonimami do napisu są wyraz oraz słowo. Rekurencyjną definicją napisu nad alfabetem T jest

1. ϵ jest łańcuchem nad alfabetem T (ϵ jest łańcuchem pustym)
2. jeśli x jest łańcuchem nad T i $a \in T$ to xa jest łańcuchem nad T
3. nic innego nie jest łańcuchem poza tym, co wynika z punktów (1) i (2)

Długość napisu oznaczana jako $|s|$, jest liczbą symboli w s. Przykładowo |napis| wynosi 5, podczas gdy |kompilator| równa się 10. Rekurencyjną definicją długości napisu s nad alfabetem T jest:

1. $|\epsilon|=0$
2. jeśli x jest łańcuchem, zaś a jest symbolem ($a \in T$), to $|ax| = 1 + |x|$

Prefiksem (ang. *prefix*) napisu s nazywa się dowolną liczbę symboli rozpoczynających napis s, a jego *sufiksem* (ang. *sufix*) dowolną liczbę symboli kończących napis s. Napis *abc* ma następujące prefiksy ϵ, *a*, *ab*, *abc*. Jego sufiksami są *c*, *bc*, *abc*, ϵ. Przedrostek lub przyrostek niepokrywający się z całym łańcuchem nazywa się *przedrostkiem właściwym* lub *przyrostkiem właściwym*.

Jeśli x i y są napisami, to złączeniem x i y, zapisywanym xy, jest napis utworzony przez dodanie y na koniec do x. Pusty napis jest elementem neutralnym złączenia, czyli $s\epsilon = \epsilon s = s$. Konkatenacja łańcuchów jest łączna $(ab)c = a(bc)$ i na ogół nie jest przemienna $ab \neq ba$.

Jeśli o łączeniu łańcuchów pomyśli się jak o iloczynie, można zdefiniować na łańcuchach również podnoszenie do potęgi.

Niech $s^0 = \epsilon$, $i>0$ oraz niech $s^i = s^{i-1}s$. Skoro $\epsilon s = s$, to $s^1 = s$, $s^2 = ss$, $s^3 = sss$...

Zbiór wszystkich słów nad alfabetem T oznacza się jako T^* i nazywa domknięciem Kleene'ego (ang. *Kleene's star*), zbiór wszystkich niepustych słów jako T^+, np.:

niech T={0,1},
wtedy $T^* = \{\epsilon, 0, 1, 00, 01, 10, 11, ...\}$,
natomiast $T^+ = \{0, 1, 00, 01, 10, 11, ...\}$.

Język (ang. *language*) jest dowolnym zbiorem napisów nad pewnym ustalonym alfabetem. Warto zaznaczyć, że definicja ta jest bardzo szeroka. Językami są np.: zbiór pusty $L=\emptyset$, zbiór zawierający pusty napis L={ϵ} oraz zbiór wszystkich zdań w języku polskim, które są poprawne gramatycznie. Definicja języka nie przypisuje żadnego znaczenia napisom w języku, zajmuje się tym *semantyka*. Formalna definicją języka jest:
Niech T będzie alfabetem, T^* – zbiorem wszystkich łańcuchów nad alfabetem T. Dowolny podzbiór L zbioru T^* nazywamy językiem L nad alfabetem T.

$$L \subseteq T^*$$

Kilka operacji zaprezentowanych na łańcuchach definiuje się również dla języków. W analizie leksykalnej przydatna jest suma, złączenie i domknięcie.
Sumę języków L i M, zapisywaną jako L∪M, definiuje się jako:

$$L \cup M = \{ s \mid s \in L \text{ lub } s \in M \}$$

Złączeniem języków L i M, które zaznacza się jako LM nazywamy:

$$LM = \{ st \mid s \in L \text{ oraz } t \in M \}$$

Domknięcie języków L i M, zapisywane jako L^* definiuje się jako:

$$L^* = \bigcup_{i=0} L^i$$

natomiast domknięcie dodatnie, oznaczane przez L^+

$$L^+ = \bigcup_{i=1} L^i$$

Dla języków można uogólnić operator podnoszenia do potęgi, definiując

$$L^0 = \{\epsilon\} \text{ oraz } L^i = L^{i-1}L.$$

3.2 Wyrażenia regularne

Naom Chomsky zdefiniował cztery klasy gramatyk oraz cztery rodzaje języków formalnych. Klasy te zostały ponumerowane od 0 do 3. Klasa zerowa równoważna jest zbiorom regularnym, które nazywa się również językami regularnymi.[8][10]

Zbiór regularny
Niech T będzie alfabetem, zbiór regularny nad alfabetem T definiujemy następująco.

1) $\varnothing$ – zbiór pusty jest zbiorem regularnym
2) $\{\varepsilon\}$ – zbiór zawierający łańcuch pusty jest zbiorem regularnym
3) $\{a \mid a \in T\}$ – zbiór zawierający łańcuch złożony z pojedynczego symbolu alfabetu jest zbiorem regularnym
4) Jeśli P i Q są zbiorami regularnymi nad alfabetem T to zbiorem regularnym jest także:
    a. $P \cup Q$ – suma teoriomnogościowa zbiorów P i Q
    b. PQ – konkatenacja (złożenie) zbiorów P i Q
    c. P^* - domknięcie Kleene'ego zbioru P
5) Nic więcej niż wynika to z punktów 1 – 4 nie jest zbiorem regularnym.

Zbiory regularne nazywa się również *językami regularnymi* (ang. *regular languages*).

Ponieważ zapis $\{\varepsilon\}\cup\{a\}\{b\}^*$ nie jest za wygodny nawet dla stosunkowo prostych struktur, zbiory regularne opisuje się wyrażeniami regularnymi. Warto zaznaczyć, że wyrażenie regularne jest uproszczonym opisem zbiorów regularnych i tylko zbiorów regularnych. Przy pomocy wyrażeń regularnych nie można opisać ani mniej ani więcej niż daje się opisać zbiorami regularnymi.

W językach programowania stosuje się wiele reguł co do wyglądu liczb, identyfikatorów lub słów kluczowych. Przykładowo, w C++ identyfikator musi zacząć się od litery lub znaku podkreślenia, po którym może wystąpić sekwencja liter lub cyfr. W teorii języków formalnych przyjmuje się, że zapis *s** oznacza „dowolną ilość znaków *s*", natomiast pionowa kreska jest operatorem alternatywy i oznacza „lub". Nawiasy służą do grupowania wyrażeń. Używając powyższej konwencji, identyfikator w języku C++ można opisać jako

litera (**litera** | **cyfra**)*

gdzie **litera** należy do zbioru [_a-zA-Z] a **cyfra** [0-9]. Wyrażenie regularne jest budowane z prostszych wyrażeń regularnych przy użyciu zestawu reguł definiowania.

Wyrażenia regularne nad alfabetem *T* definiujemy następująco:

1) $\varnothing$ jest wyrażeniem regularnym oznaczającym zbiór pusty będący zbiorem regularnym
2) ε jest wyrażeniem regularnym oznaczającym zbiór zawierający napis pusty $\{\varepsilon\}$ będący zbiorem regularnym
3) a jest wyrażeniem regularnym oznaczającym zbiór $\{a \mid a \in T\}$ zawierający łańcuch złożony z pojedynczego symbolu alfabetu będący zbiorem regularnym
4) jeśli p i q są wyrażeniami regularnymi oznaczającymi odpowiednio zbiory regularne P i Q nad T to wyrażeniami regularnymi są także:
    a. $p|q$ – wyrażenie regularne oznaczające $P\cup Q$ – sumę teoriomnogościową zbiorów P i Q będącą zbiorem regularnym
    b. pq – wyrażenie regularne oznaczające PQ – złożenie (konkatenację) zbiorów P i Q będące zbiorem regularnym
    c. p^* - wyrażenie regularne oznaczające P^* - domknięcie Kleene'ego zbioru P będące zbiorem regularnym
5) nic innego poza tym co wynika z punktów 1 – 4 nie jest wyrażeniem regularnym.

W Tabela 2 przedstawiono najczęściej spotykane aksjomaty wraz z ich słownym opisem.

Tabela 2 Prawa algebraiczne dla wyrażeń regularnych

AKSJOMAT	OPIS
$r\|s = s\|r$	operator | (alternatywy) jest przemienny
$r\|(s\|t) = (r\|s)\|t$	operator | (alternatywy) jest łączny
$r(st) = (rs)t$	złączenie jest łączne
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	rozdzielność złączenia względem alternatywy
$\varepsilon r = r\varepsilon = r$	ε jest elementem neutralnym złączenia
$r^* = (\varepsilon\|r)^*$	relacja między ε a *
$r^{**} = r^*$	* jest idempotentna
$\varnothing^* = \varepsilon$	relacja pomiędzy $\varnothing$ a ε

Tabela 3 Liczby w C++ zapisane przy pomocy wyrażeń regularnych

SYMBOL LEKSYKALNY	WYRAŻENIE REGULARNE
cyfra	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
cyfry	**cyfra cyfra***
opcjonalnyUlamek	.**cyfry** | ε
opcjonalnyWykladnik	((E | e) (+ | – | ε) **cyfry**) | ε
liczba	**cyfry opcjonalnyUlamek opcjonalnyWykladnik**

Niektóre konstrukcje pojawiają się w wyrażeniach regularnych na tyle często, że wprowadzono dla nich skróty notacyjne. Zostały one zaprezentowane w Tabela 4.

Tabela 4 Skróty używane przy zapisane wyrażeń regularnych z ich wyjaśnieniem i przykładem

SKRÓT NOTACYJNY	ZNACZENIE I PRZYKŁAD
+	„co najmniej jedno wystąpienie”, a+ = {a | aa | aaa | aaaa | aaaaa...}
?	„opcjonalnie”, a? = { ε | a }
[*znaki*]	„klasa znaków”, [0-9] = { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
[^*znaki*]	„klasa znaków nie należąca do *znaki*”, [^123] = { 4 | 5 | 6 | 7 | 8 | 9 }
a{*n*,*m*}	„znak a powtórzony od *n* do *m* razy”, a{2,3} = { aa | aaa }

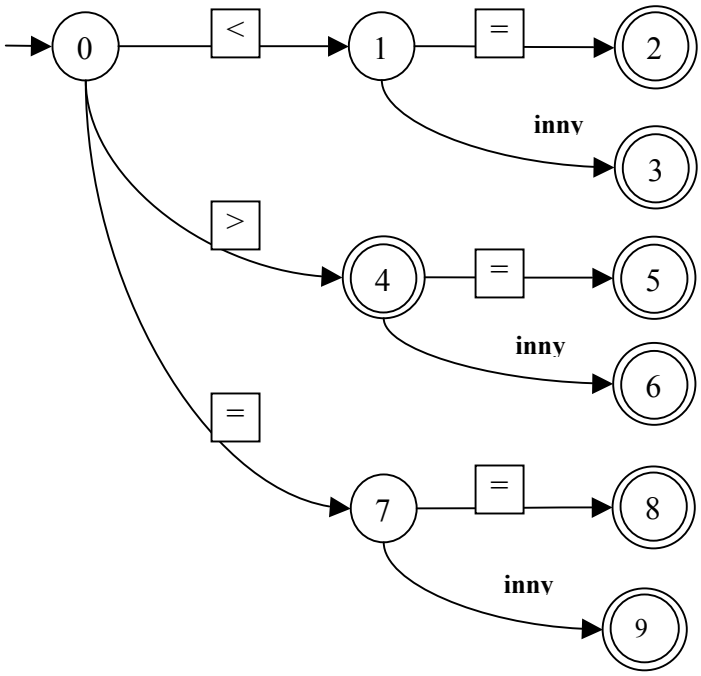
Wyrażenia regularne mogą jedynie opisać ustaloną liczbę powtórzeń albo dowolną liczbę powtórzeń danej konstrukcji. Wyrażenia regularne nie mogą być użyte do opisania struktur zrównoważonych i zagnieżdżonych. Wyrażeniami regularnymi nie można zatem opisać np.: *napisów Hollerintha* o postaci $nHa_1a_2...a_n$ pochodzących z wczesnych wersji Fortranu. Podobnie napisów w postaci { wcw | $w \in T^+$, T={a,b}} jak również zbiór wszystkich napisów zawierających poprawnie zagnieżdżone nawiasy nie można opisać wyrażeniami regularnymi.

3.3 Automaty stanowe skończone

Automat skończony (ang. *finite state machine/automata*) jest modelem matematycznym systemu o dyskretnym wejściu i dyskretnym wyjściu. System taki może znajdować się w jednej z możliwych konfiguracji, czyli stanie. Stan systemu stanowi podsumowanie wiedzy dotyczącej poprzednich wejść, która jest potrzebna do określenia zachowania systemu po następnych wejściach.

Właściwości automatów skończonych sprawiają, że nadają się one znakomicie do rozpoznawania wzorców leksykalnych. Jako wejście podajemy sekwencję liter, jako wyjście otrzymujemy, czy i jaki napis został rozpoznany.

Na Rys. 3.1 zaprezentowano prosty automat skończony. Składa się on z 11 stanów ponumerowanych od *0* do *10*. Stan *0* jest stanem początkowym. Reprezentuje on maszynę stanową na początku działania, maszynę która nie otrzymała jeszcze żadnych danych wejściowych. Stany *2*, *3*, *4*, *5*, *6*, *7*, *9*, i *10* są stanami końcowymi – akceptującymi. Stan akceptujący jest takim stanem w którym maszyna rozpoznała podane wejście jako poprawne. Pomiędzy stanami są przejścia oznaczone strzałkami. Każde przejście jest etykietowane. W naszym przypadku etykietą jest symbol należący do alfabetu T = {<,>, =} oraz *symbol terminalny* **inny**.



Rys. 3.1 Automat skończony rozpoznający operatory logiczne: <, <=, =, ==, >, >=[2]

Dla automatu skończonego z Rys. 3.1 oraz danych wejściowych >= automat znajdzie się w stanie *8* po znaku > oraz w stanie *9* po znaku =. Oznacza to, że napis >= należy do języka akceptowanego przez ten automat oraz jest poprawnym napisem w tym języku. Napis <8 spowoduje rozpoznanie operatora mniejszości, i spowoduje przejście maszyny przez stan *1* do stanu *3* oraz jednego nieznanego symbolu. Symbol * przy stanie akceptującym oznacza, że został wczytany o jeden symbol za dużo i trzeba go zwrócić do bufora z nieoczytanymi jeszcze znakami.

Formalną definicją automatu skończonego jest
Automatem skończonym nazywamy uporządkowaną piątkę $A = < T, Q, F, q_o, \delta >$ gdzie:
T – zbiór symboli z alfabetu wejściowego
Q – zbiór stanów, $\#Q<\infty$
F – zbiór stanów końcowych, $F \subseteq Q$
q_o – stan początkowy, $q_o \in Q$
δ – funkcja przejścia, $\delta: Q \times (T \cup \{\varepsilon\}) \rightarrow 2^Q$

Język L jest akceptowany przez automat A (co oznaczamy $L(A)$) $\Leftrightarrow$

$$L = L(A) = \{ x \in T^* \mid x \text{ jest akceptowane przez } A \}$$

$x \in T$ jest słowem akceptowalnym przez automat $A \Leftrightarrow$

$$(\exists q \in F)((q_o, x) \rightarrow^* (q, \varepsilon))$$

Konfiguracją blokującą nazywamy taką parę (q, w), że

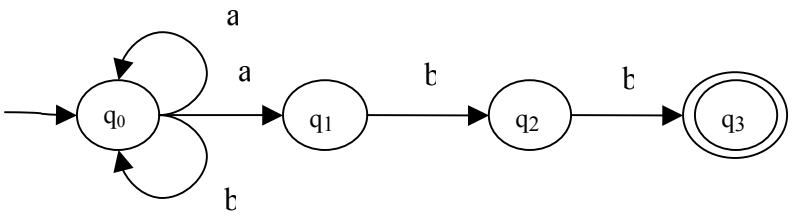
$$(q, w) \Leftrightarrow \neg (\exists(q', w')) ((q, w) \rightarrow^* (q', w'))$$

gdzie $w \in T^*$ i jest nieprzeczytaną częścią napisu wejściowego.

Przykład 3.1 [2]
Język regularny opisany wyrażeniem regularnym (a | b)* abb można opisać formalnie
T = {a,b}
Q = {q_o, q_1, q_2, q_3}
F = {q_3}
q_0 – stan początkowy
δ – funkcja przejścia

STAN	a	b
q_0	{q_0, q_1}	{q_0}
q1	∅	{q_2}
q2	∅	{q_3}

Rys. 3.2 Funkcja przejścia dla wyrażenia regularnego (a | b)* abb



Rys. 3.3 Diagram przejść dla wyrażenia regularnego (a | b)* abb

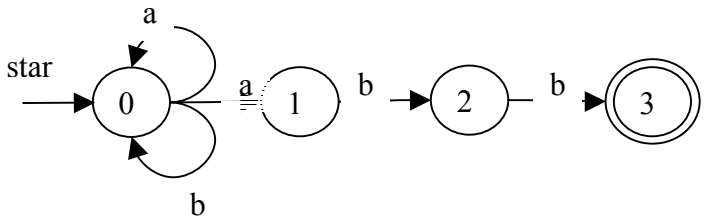
Na rysunku stany akceptujące oznacza się podwójnym okręgiem.□

Z praktycznego punktu widzenia, najważniejszą grupą automatów skończonych są *deterministyczne automaty skończone* oznaczana skrótem DAS. Ich formalna definicja narzuca dodatkowe ograniczenie w stosunku do zwyczajnych, niedeterministycznych automatów skończonych NAS.

Automat skończony jest deterministyczny wtedy i tylko wtedy, gdy:

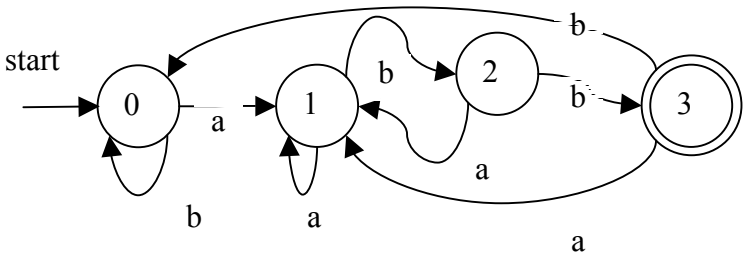
$$(\forall Q \in q)(\#\delta(q, \varepsilon) = 0)$$
$$(\forall a \in T)(\forall q \in Q)(\# \delta(q, a) \leq 1)$$

Deterministyczny automat skończony ma co najwyżej jedno przejście z każdego stanu dla ustalonego symbolu oraz nie posiada przejść dla napisów pustych. Największą zaletą deterministycznego automatu jest to, że po wczytaniu konkretnej sekwencji znaków jesteśmy pewni w którym stanie jest automat.



Rys. 3.4 Przykład niedeterministycznego automatu dla wyrażenia (a|b)abb

Zaletą automatów niedeterministycznych jest ich prostota projektowania, wadą złożoność programu komputerowego akceptującego język równoważny do języka akceptowanego przez automat. Istnieje jednak algorytm umożliwiający zamianę automatu niedeterministycznego, na równoważny deterministyczny. Dzięki temu, można najpierw w prosty sposób zaprojektować niedeterministyczny automat dla języka L, a następnie zamienić go na deterministyczny automat który w bardzo prosty sposób daje wyrazić się w językach programowania.



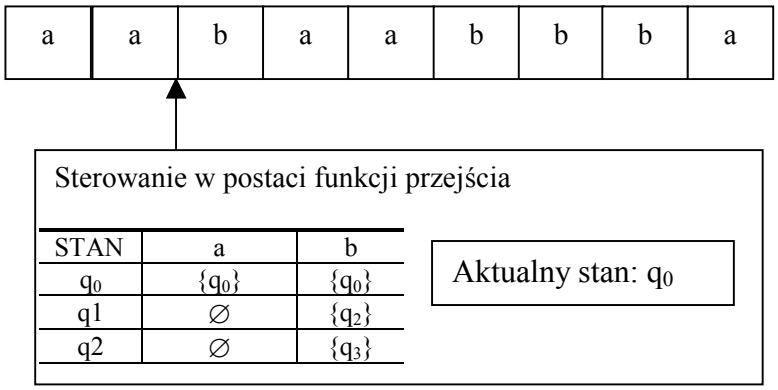
Rys. 3.5 Automat deterministyczny [2]

Szczegółowe omówienie algorytmu konstrukcji podzbiorów przedstawione w książce [2].

Nazwa automat pochodzi od bardzo luźnego skojarzenia. Można sobie wyobrazić, że DAS jest „pseudourządzeniem" dysponującym taśmą wejściową, na której zapisany jest badany łańcuch i nad którą może poruszać się głowica odczytująca pojedyncze symbole. Głowica ma tylko możliwość ruchu w prawo, tzn. od początku łańcucha ku jego końcowi. Każdorazowo głowica przesuwa się tylko o jeden symbol. W każdym kroku głowica jest w stanie przeczytać tylko jeden symbol $t \in T$ na taśmie, a automat może być opisany przez jeden ze stanów ze skończonego ich zbioru Q. Automat posiada swoje sterowanie opisane przez funkcję przejścia δ, która na podstawie aktualnego stanu i symbolu odczytanego przez głowicę określa następny stan w następnym kroku. Początkowo głowica ustawiona jest na pierwszym symbolu badanego słowa a sam automat znajduje się w stanie początkowym q_o. Każdy krok pracy automatu składa się z odczytania przez głowicę symbolu z taśmy, określenia nowego stanu i przesunięcia głowicy o jeden symbol w prawo na taśmie wejściowej. Kroki są powtarzane dopóki spełnione są warunki

1. automat znajduje się w stanie nie należącym do zbioru stanów akceptujących F
2. nie zostały wczytane wszystkie symbole z taśmy
3. dla aktualnego stanu automatu oraz symbolu na taśmie zdefiniowany jest przez funkcję przejścia nowy stan.

Zatrzymanie automatu w stanie należącym do zbioru stanów akceptujących oznacza, że napis zapisany na taśmie jest akceptowany przez automat, natomiast zatrzymanie automatu w stanie nie akceptującym oznacza brak akceptacji napisu.



Rys. 3.6 Schemat automatu skończonego

3.4 Automat skończony ze stosem

Automat skończony ze stosem jest automatem skończonym, wyposażonym w sterowanie zarówno taśmą jak i stosem. Automat taki posiada taśmę wejściową, sterowanie skończone oraz stos. Stos jest łańcuchem symboli pewnego alfabetu. Niedeterministycznym automatem skończonym ze stosem nazywamy siódemkę uporządkowaną $A = < T, Q, F, q_o, S, s_0\ \delta >$ gdzie

T – zbiór symboli z alfabetu wejściowego
Q – zbiór stanów, $\#Q<\infty$
F – zbiór stanów końcowych, $F \subseteq Q$
q_o – stan początkowy, $q_o \in Q$
S – zbiór symboli stosowych
s_0 – szczególny symbol stosowy zwany stosowym symbolem początkowym $s_0 \in S \cup \{\varepsilon\}$

δ – funkcja przejścia, δ: $Q \times (T \cup \{\varepsilon, \$\}) \times S^* \rightarrow 2^{Q \times S^*}$, \$ ogranicznik końca słowa wejściowego

Automat ze stosem może być również maszyną deterministyczną, jeżeli dodamy następujące ograniczenia:

1. $(\forall q \in Q)\ (\forall a \in T \cup \{\varepsilon, \$\})\ (\forall \gamma \in S^*)\ (\#\delta(q, a, \gamma) \leq 1)$
2. $(\delta(q, a, \alpha) \neq \phi \wedge \delta(q, a, \beta) \neq \phi \wedge \alpha \neq \beta) \Rightarrow$ żaden z łańcuchów α oraz β nie jest przyrostkiem drugiego łańcucha
3. $(\delta(q, a, \alpha) \neq \phi \wedge \delta(q, a, \beta) \neq \phi) \Rightarrow$ żaden z łańcuchów α oraz β nie jest przyrostkiem drugiego łańcucha

Język L jest akceptowany przez automat ze stosem A (co oznaczamy $L(A)$) $\Leftrightarrow$

$$L = L(A) = \{x \in T^* \}\ x \text{ jest akceptowane przez } A\}$$

$x \in T^*$ jest słowem akceptowanym przez automat A ze stosem $\Leftrightarrow$

$$(\exists q \in F)\ ((s_0, q_0, x\$) \Rightarrow^* (\varepsilon, q, \$))$$

3.5 Maszyna Turinga

Podstawowy model maszyny Turinga ma skończone sterowanie, taśmę wejściową podzieloną na komórki oraz głowicę taśmy, mogącą obserwować w dowolnej chwili tylko jedną komórkę taśmy. Taśma ta jest prawostronnie nieskończona. Każda z komórek taśmy może zawierać dokładnie jeden ze skończonej liczby symboli taśmowych. Na początku, pierwsze *n* komórek od lewej (gdzie n≥0 jest pewną skończoną liczbą) zawiera wejście, będące łańcuchem symboli wybranych z podzbioru symboli taśmowych, zwanego zbiorem symboli wejściowych. Wszystkie pozostałe komórki, których jest nieskończenie wiele, zawierają symbol pusty. Nie traktuje się go jako symbolu wejściowego.
W zależności od symbolu obserwowanego przez głowicę taśmy oraz stanu sterowania skończonego, maszyna w pojedynczym ruchu:
1. zmienia stan
2. drukuje symbol w obserwowanej komórce taśmy, nadpisując poprzedni symbol
3. przesuwa głowicę o jedną komórkę w lewo lub w prawo.

Warto zauważyć, że maszyna Turinga ma możliwość pisania na taśmie.

Formalnie maszyna Turinga jest siódemką uporządkowaną $A = < T, Q, U, F, q_o, B, \delta >$

T – podzbiór U niezawierający symbolu B
Q – zbiór stanów, $\#Q<\infty$
U – symbole taśmowe, skończony zbiór symboli
F – zbiór stanów końcowych, $F \subseteq Q$
q_o – stan początkowy, $q_o \in Q$
B – znak pusty (ang. *blank*), $B \in U$
δ – funkcja przejścia, δ: $Q \times U \rightarrow 2^{Q \times U \times \{\mathbf{L}, \mathbf{P}\}}$

gdzie, **L** i **P** oznaczają odpowiedni ruch w lewo i ruch w prawo.

3.6 Automat liniowo ograniczony

Automatem liniowo ograniczonym nazywamy niedeterministyczną maszynę Turinga spełniającą dwa warunki:
1. Alfabet wejściowy zawiera dwa specjalne symbole % i $, będące odpowiednio lewym i prawym znacznikiem końca.
2. Automat ten nie ma ruchów na lewo od znaku % i na prawo od znaku $. Nie może on również nadpisać symbolu % i $.

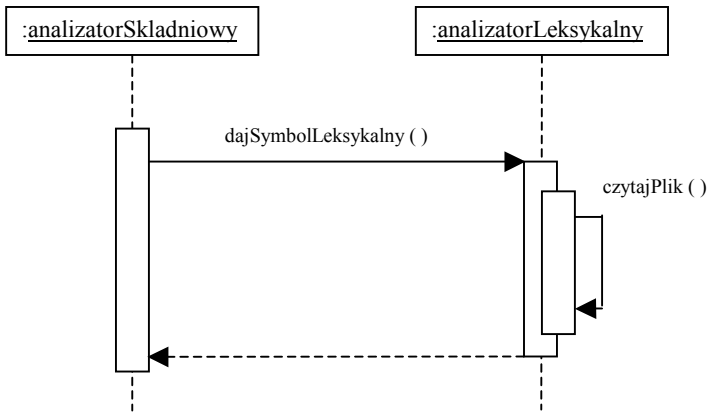
Automat liniowo ograniczony jest maszyną Turinga, która nie dysponuje nieskończoną taśmą do obliczeń. Musi się ona ograniczyć do części tej taśmy zawartej pomiędzy znakami % i $.
Formalnie automat liniowo ograniczony jest uporządkowaną ósemką $A = < T, Q, U, F, q_o, \%, \$, \delta >$

T – podzbiór U niezawierający symbolu B
Q – zbiór stanów, $\#Q<\infty$
U – symbole taśmowe, skończony zbiór symboli
F – zbiór stanów końcowych, $F \subseteq Q$
q_o – stan początkowy, $q_o \in Q$
$\%$ – lewy terminator taśmy, znajduje się cały czas na taśmie, nie jest częścią napisu
$\$$ – prawy terminator taśmy, znajduje się cały czas na taśmie, nie jest częścią napisu
δ – funkcja przejścia, δ: $Q \times U \rightarrow 2^{Q \times U \times \{\mathbf{L}, \mathbf{P}\}}$

gdzie, **L** i **P** oznaczają odpowiedni ruch w lewo i ruch w prawo.

3.7 Zasada działania analizatora składniowego

Zdania w języku składają się z ciągów symboli leksykalnych. Każdy symbol leksykalny składa się z sekwencji pewnej liczby znaków. Zadaniem analizatora leksykalnego (ang. *scaner*) jest wyodrębnienie we wczytywanym tekście symboli leksykalnych, a następnie przekazanie ich do analizatora składniowego.



Rys. 3.7 Diagram współpracy analizatorów składniowego i leksykalnego, para typu producent – konsument

Analizator (ang. *scanner*) wczytuje znaki z wejścia, grupuje je w symbole leksykalne i przekazuje te symbole do dalszych etapów kompilacji. W niektórych sytuacjach skaner musi wczytać kilka znaków na przód zanim zdecyduje, jaki symbol ma być przekazany dalej. Na przykład, skaner po trafieniu na znak < musi wczytać kolejny symbol, aby upewnić się, czy znak < jest początkiem operatora mniejszy lub równy czy też może jest jednoznakowym operatorem mniejszości. Jeżeli kolejny symbol po znaku < jest różny od znaku =, okazało się, że w tekście wystąpił operator mniejszości i został wczytany jeden znak za dużo. Znak ten trzeba będzie zwrócić na wejście, ponieważ prawdopodobnie jest początkiem kolejnego symbolu leksykalnego. Żeby przyśpieszyć tę operację, przeważnie stosuje się dla skanera bufor wejściowy i wyjściowy.

Skaner jest częścią kompilatora czytającą tekst źródłowy, zatem może on wykonywać pewne dodatkowe zadania związane z interfejsem użytkownika. Takimi zadaniami może być pomijanie komentarzy, białych znaków i znaków nowego wiersza. Innym zadaniem może być zliczanie wczytanych linii, dzięki temu kompilator będzie w stanie poinformować użytkownika, w której linii znaleziony został błąd. Również makra preprocesora mogą zostać rozwinięte w tej fazie.

W analizie leksykalnej nazwy *symbol leksykalny*, *wzorzec* i *leksem* mają specyficzne znaczenia. Ogólnie ten sam symbol leksykalny może zostać wygenerowany z całego zbioru różnych ciągów znaków wejściowych. Zbiór ten opisuje reguła zwana wzorcem. Leksem jest sekwencją pasującą do wzorca tego symbolu leksykalnego. W Tabela 5 przedstawiono popularne symbole leksykalne wraz z leksemami pasującymi do nich a także opis symbolu w języku regularnym.

Tabela 5 Przykłady symboli leksykalnych

SYMBOL LEKSYKALNY	PRZYKŁADOWY LEKSEM	OPIS WZORCA
const	const	const
for	for	for
operatorRelacji	<,<=,==,!=,=>,>	< | <= | == | != | => | >
identyfikator	zmienna3, _a, i, rozmiar	[_a-zA-Z]([_a-zA-Z]|[0-9])*
cyfraHeksadecymalna	0X324, 0xffff	0[Xx]([0-9]|[a-f]|[A-F])+

Symbole leksykalne są traktowane jak *symbole terminalne gramatyki* języka źródłowego. Zgodnie z notacją ich nazwy pisze się pogrubioną czcionką.

Jak już zostało wspomniane, symbol leksykalny opisuje zbiór napisów pasujących do wzorca. Skaner zwraca analizatorowi składniowemu, że znalazł **operatorRelacji**, **identyfikator** lub **cyfrę**, ale dla dalszych etapów kompilacji nie mniej ważne jest, jaki to był operator, jaki identyfikator i jaką wartość ma rozpoznana cyfra. Dlatego niektóre symbole leksykalne mają swoje wartości zwane niekiedy atrybutami.

Dla deklaracji zmiennej w języku C++, `const double PI = 3.1416;` skaner zwróci parserowi sekwencje par <**terminal**, wartość>: <**keyword**, const>, <**keyword**, double>, <**identyfikator**, PI>, <**operator**, =>, <**liczba**, 3.1416>. Terminale i ich przykładowe wartości przedstawiono w tabeli 5.

Tabela 6 Symbole terminalne wraz z ich przykładowymi wartościami

SYMBOL TERMINALNY	ATRYBUT SYMBOLU
keyword	const
keyword	double
identyfikator	PI
operator	=
liczba	3.1416

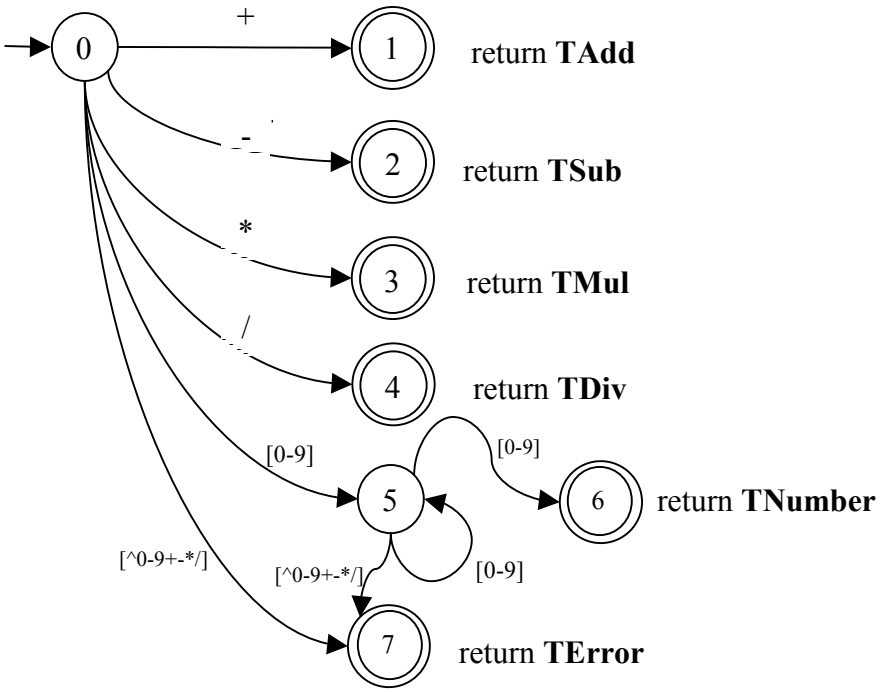
W większości języków programowania, symbolami leksykalnymi są słowa kluczowe, operatory, identyfikatory, stałe, literały znakowe, znaki przestankowe takie jak nawiasy, przecinki, średniki… W wielu językach niektóre ciągi znaków są zarezerwowane, tzn. ich znaczenie jest wstępnie zdefiniowane i nie może być zmieniane przez użytkownika. Takie ciągi znaków nazywamy *słowami kluczowymi*. Jeżeli słowo kluczowe nie jest zarezerwowane, skaner musi odróżniać słowa kluczowe od identyfikatorów zdefiniowanych przez użytkownika. Przykładem języka w którym słowa kluczowe nie są zarezerwowane jest PL/I. Ilustruje to poniższy poprawnie działający przykład [2]

IF THEN THEN THEN = ELSE ; ELSE ELSE = THEN ;

Ogólna zasada działania skanera opiera się na implementacji deterministycznego automatu skończonego w dowolnym wysokopoziomowym języku programowania. Poniżej przedstawiono przykład automatu skończonego w pseudokodzie.

Podczas implementowania analizatora leksykalnego, najprościej użyć instrukcji `switch case`. Każda opcja `case` odpowiada wtedy jednemu stanowi maszyny stanowej. Globalna zmienna Atrybut przechowuje wartość symbolu leksykalnego, a typ wyliczeniowy Terminal definiuje możliwe do zwrócenia przez skaner rozpoznawalne symbole leksykalne. Ponadto, wykorzystano pomocniczą funkcję, `getChar`(), która przekazuje kolejny znak z wejścia oraz `putChar`(), która zwraca wczytany znak do strumienia wejściowego.

Warto przypomnieć, że znak * przy stanie akceptującym oznacza, że został wczytany jeden znak za dużo, znak który jest początkiem innego leksemu i znak ten należy zwrócić na wejście.



Rys. 3.8 Przykład deterministycznego automatu skończonego rozpoznającego operatory logiczne

```
enum Terminal{
      Terror
      , Tadd
      , Tsub
      , Tmul
      , Tdiv
      , Tnumber
};

char getChar(){
      // funkcja zwraca kolejny znak z wejścia

}

void putChar(){
      /*
      funkcja zwraca na wejście odczytany znak,
      sekwencja: {getChar(); putChar();} nie zmienia wejścia
      */
}

int Atrybut; // zmienna przechowuje wartość rozpoznanej liczby
```

Rys. 3.9 Typy oraz pomocnicze funkcje użyteczne do napisania analizatora leksykalnego

```
Terminal getLexicalSymbol(){
      char currentChar = ``;  // obecnie wczytany znak
      int currentState = 0;   // w którym stanie obecnie jest maszyna
      std::string number= "";// przechowuje cyfry rozpoznanej licz.

      while( true ){
            currentChar = getChar();
            switch( currentState){
                  case 0:// reprezentuje stan 0 maszyny stanowej
                        if( currentChar == '+')
                              currentState = 1;
                        else if( currentChar == '-')
                              currentState= 2;
                        else if( currentChar == '*')
                              currentState= 3;
                        else if( currentChar == '/')
                              currentState = 4;
                        else if( isdigit(currentChar){
                              number += currentChar;
                              currentState= 5;
                        }
                        else
                              currentState= 7;
                        break;
                  case 1: return TAdd;
                  case 2: return TSub;
                  case 3: return TMul;
                  case 4: return TDiv;
                  case 5:
                        if( isDigit(currentChar)) {
                              number += currentChar;
                              currentState= 5;
                        }
                        else if( currentChar== '+'
                                    or currentChar== '-'
                                    or currentChar== '*'
                                    or currentChar== '/'
                        )
                              currentState= 6;
                        else
                              currentState= 7;
                              break;
                  case 6:
                        putBack();
                        Atrybut = atoi(number.c_str());
                        return TNumber;
                  case 7:
                        return TError;
            }
      }
};
```

Rys. 3.10 Pseudokod reprezentujący maszynę stanową z Rys. 3.8

4 Analizy składniowa

Zadaniem analizy syntaktycznej, jest sprawdzenie, czy ciąg symboli leksykalnych zwróconych przez lekser jest zgodny z gramatyką. Symbole leksykalne są grupowane w coraz większe i większe wyrażenia gramatyczne zgodnie z regułami gramatycznymi. Wyrażenia gramatyczne są zazwyczaj reprezentowane przez hierarchiczną strukturę danych, jaką może być *drzewo wyprowadzenia*.
Dla każdej gramatyki bezkontekstowej można zbudować analizator składniowy o złożoności czasowej $O(n^3)$, gdzie n jest długością analizowanego napisu. Dla celów praktycznych jest to jednak złożoność za czasochłonna. Projektując gramatyki języków programowania, które będą używane w przemyśle nadaje się im specjalną postać, dzięki czemu algorytmy parsujące zyskują złożoność czasową $O(n)$.
Istnieje wiele metod analizy syntaktycznej, jednak największą popularnością cieszą się metoda zstępująca (ang. *top-down*) i metoda wstępująca (ang. *bottom-up*). Określenia te odnoszą się do sposobu budowania drzewa wyprowadzenia. W metodzie zstępującej drzewo budowane jest od korzenia w kierunku liści natomiast w metodzie wstępującej na samym początku utworzymy liście drzewa i łącząc je przy pomocy węzłów wewnętrznych zmierzamy w kierunku korzenia. Analizatory wstępujące są trudniejsze do zaimplementowania, mogą być jednak używane do szerszej klasy gramatyk.

4.1 Gramatyka

Wyraz *gramatyka* pochodzi z języka greckiego i oznacza naukę, której obiektem badań jest język. W potocznej mowie, wyrazu gramatyka często używa się jednak nie w znaczeniu nauki, ale zbioru reguł służących do opisania języka.

Gramatyką nazywamy czwórkę uporządkowaną $G = <N, T, P, S>$ gdzie:

N – zbiór symboli nieterminalnych
T – zbiór symboli terminalnych
P – zbiór produkcji, z których każda ma postać $\alpha \rightarrow \beta$
$S \in N$ – symbol startowy gramatyki, wyróżniony symbol początkowy (nieterminal)

przy czym

$\#N<\infty, \#T<\infty, \#P<\infty$
$N \cap T = \varnothing$
$P \subseteq (N \cup T)^{+} \times (N \cup T)^{*}$
$P = \{ \alpha \rightarrow \beta \mid \alpha \in (N \cup T)^{+}, \beta \in (N \cup T)^{*} \}$

Terminale są symbolami podstawowymi, z których tworzone są napisy. Słowo *symbol leksykalny* jest synonimem terminala. Przykładami terminali z języka C++ są: **const**, **template**, **identyfikator**, **liczbaRzeczywista**. Konkretnymi egzemplarzami terminali są leksemy. Terminal **identyfikator** może mieć następujące leksemy: *i*, *zmienna*, *licznik*, *imie*…

Nieterminale są zmiennymi składniowymi opisującymi zbiory napisów. Narzucają one językowi strukturę hierarchiczną, która jest przydatna zarówno w analizie składniowej, jak i w translacji.

Jeden z symboli nieterminalnych w gramatyce jest wyróżniony jako *symbol startowy*, a zbiór napisów, które on definiuje, jest *językiem* definiowanym przez tę gramatykę.

Produkcje gramatyki specyfikują sposób, w jaki terminale i nieterminale mogą być łączone w napisy. Każda produkcja składa się z nieterminala, a następnie strzałki (zastępczo można użyć znaku ::=) i napisu złożonego z terminali i nieterminali. Część produkcji przed strzałką nazywana jest lewą stroną produkcji, a część produkcji po strzałce jej prawą stroną.

Przykład 4.1
Instrukcję warunkową można opisać następującą produkcją:

instrukcjaWarunkowa → **if (** *wyrazenie* **)** *instrukcjaZlozona* **else** *instrukcjaZlozona*

Terminalami we powyższym wzorze są **if**, **(**, **)** oraz **else**. Nieterminalami są *instrukcjaWarunkowa*, *wyrazenie* i *instrukcjaZlozona*.□

Na Rys. 4.1 przedstawiono gramatykę dla wyrażeń matematycznych. Gramatyka ta uwzględnia priorytet operatorów. Na samym początku zostaną obliczone wartości w nawiasach, następnie mnożenie oraz dzielenie i w końcu dodawanie oraz odejmowanie. Im wyższy priorytet ma operator niż „niżej" jest w gramatyce.

wyrazenie → *wyrazenie* **+** *skladnik*
wyrazenie → *wyrazenie* **-** *skladnik*
wyrazenie → *skladnik*
skladnik → *skladnik* ***** *czynnik*
skladnik → *skladnik* **/** *czynnik*
skladnik → *czynik*
czynnik → **liczba | (** *wyrazenie* **)**

Rys. 4.1 Gramatyka dla wyrażeń matematycznych z uwzględnieniem operatorów +,-,*,/,()

W gramatyce na Rys. 4.1, terminalami są *, /, +, –, (,) i **liczba**, Nieterminalami są *wyrazenie*, *skladnik* i *czynnik*, ponadto *wyrazenie* jest symbolem startowym. Zapis użyty w produkcji *czynnik* → **liczba | (liczba)** jest skrótem notacyjnym dla dwóch produkcji *czynnik* → **liczba** oraz *czynnik* → **(liczba)**. Notacje alternatywy można uogólnić dla *n* produkcji.

Jeżeli w gramatyce *G*, bezpośrednio ze słowa ω można wyprowadzić słowo ψ, mówimy, że słowo ψ jest bezpośrednio wyprowadzalne ze słowa ω i zapisujemy to jako $\omega \Rightarrow_G \psi$.

Słowo ψ, jest wyprowadzalne z ω, jeżeli:
$\omega = \gamma\alpha\delta$
$\psi = \gamma\beta\delta$
$(\alpha \rightarrow \beta) \in P$
$\alpha, \beta, \gamma, \delta, \psi, \omega \in (N \cup T)^*$

Słowo ψ jest wyprowadzalne ze słowa ω w gramatyce *G*, co zapisujemy

$$\omega \Rightarrow_G^+ \psi$$

jeżeli istnieją $\varphi_0, \varphi_1, \ldots, \varphi_n \in (N \cup T)^*$ takie, że:
$\varphi_0 = \omega$
$\varphi_n = \psi$
$\varphi_{i-1} \Rightarrow_G \varphi_i$, dla i=1, 2, …, n.
Sekwencję $\varphi_0, \varphi_1, \ldots, \varphi_n$ nazywamy wyprowadzeniem o długości *n*.

Ponadto definiuje się

$$(\omega \Rightarrow_G^* \psi) \Leftrightarrow (\omega \Rightarrow_G^+ \psi) \vee (\omega = \psi)$$

Gramatyka jest jednym ze sposobów definiowania języka formalnego. Mając daną gramatykę G oznaczamy przez $L(G)$ zbiór wszystkich słów, które mogą być w tej gramatyce wyprowadzone z symbolu początkowego Z. Zbiór ten nazywamy językiem generowanym przez daną gramatykę.

$$L(G) = \{ x \in T^* \mid S \Rightarrow_G^* x \}$$

Łańcuch $x \in (N \cup T)^*$ nazywamy *formą zdaniową* gramatyki G, jeżeli można go wyprowadzić z symbolu początkowego Z.

$$x \in (N \cup T)^* \text{ jest } \textit{formą zdaniową} \Leftrightarrow S \Rightarrow_G^* x$$

Forma zdaniowa, która nie zawiera nieterminali nazywana jest *zdaniem*.

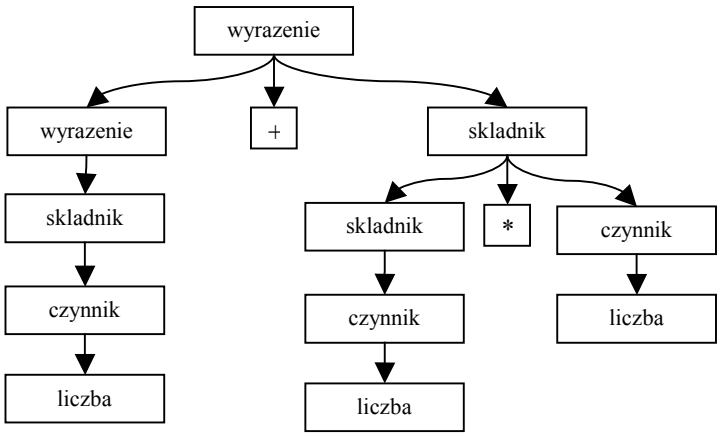
Według gramatyki z przykładu 4.2 napis **liczba + liczba * liczba** jest zdaniem, ponieważ jest możliwe wyprowadzenie *wyrazenie* $\Rightarrow^+$ **liczba + liczba * liczba.**

Przykład 4.2

Wyprowadzenie napisu **liczba + liczba * liczba** z symbolu startowego gramatyki.

.wyrazenie $\Rightarrow$
wyrazenie + *skladnik* $\Rightarrow$
wyrazenie + *skladnik* * *czynnik* $\Rightarrow$
wyrazenie + *skladnik* * **liczba** $\Rightarrow$
wyrazenie + *czynnik* * **liczba** $\Rightarrow$
wyrazenie + **liczba * liczba** $\Rightarrow$
skladnik + **liczba * liczba** $\Rightarrow$
czynnik + **liczba * liczba** $\Rightarrow$ **liczba** + **liczba * liczba** □

Na Rys. 4.2 zaprezentowano drzewo składniowe dla wyprowadzenia z Przykład 4.2. *Drzewem wyprowadzenia* nazywamy graficzną reprezentację wyprowadzenia, w której najczęściej pomija się kolejność zmian. Korzeń tego drzewa jest symbolem startowym gramatyki, każdy węzeł wewnętrzny reprezentuje nieterminal A, natomiast potomkowie tego węzła, od lewej do prawej, oznacza się kolejnymi symbolami z prawej strony produkcji za pomocą której nieterminal A został umieszczony w drzewie. Liście w drzewie wyprowadzenia mogą być oznaczane przez nieterminale lub terminale, a czytane od lewej do prawej tworzą formę zdaniową zwaną *plonem* tego drzewa.

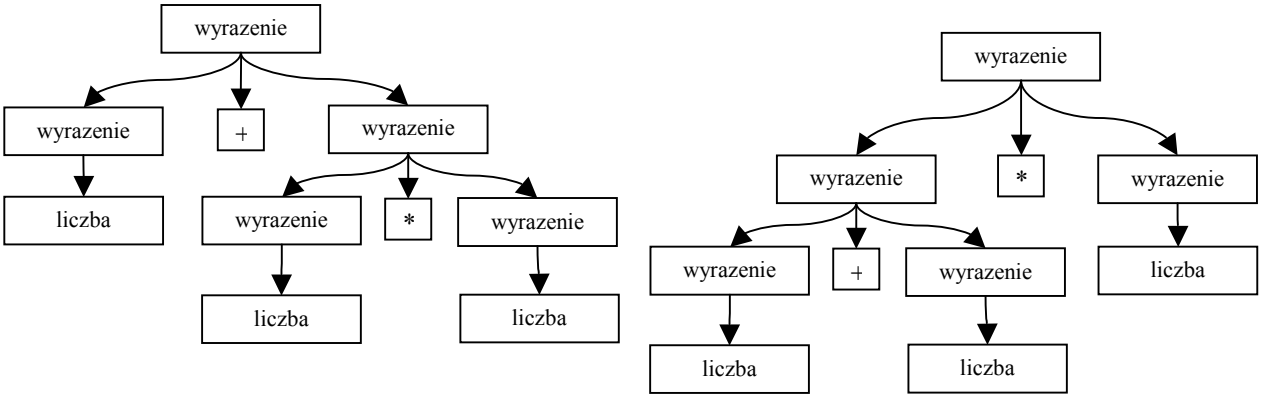


Rys. 4.2 Drzewo wyprowadzenia dla napisu **liczba + liczba * liczba**

Gramatyka, w której dane zdanie ma więcej niż jedno drzewo wyprowadzenia nazywana jest *niejednoznaczną*. Przykładem takiej gramatyki jest gramatyka zaprezentowana na rysunku 4.2. Aby wykazać, że gramatyka jest niejednoznaczna, wystarczy znaleźć dwa różne drzewa dla tego samego napisu. Na rysunku 4.4 przedstawiono takie drzewa dla napisu **liczba** + **liczba** * **liczba**. Niejednoznaczność wynika z tego, że operatory multiplikatywne są na tym samym poziomie struktury hierarchicznej co addytywne. Skutkiem tego jest to, że wyrażenie **liczba** + **liczba** * **liczba** może być sparsowane i obliczone jako (**liczba** + **liczba**) * **liczba** lub **liczba** + (**liczba** * **liczba**).

wyrazenie → *wyrazenie* + *skladnik*
wyrazenie → *wyrazenie* – *składnik*
wyrazenie → *wyrazenie* * *składnik*
wyrazenie → *wyrazenie* / *składnik*
wyrazenie → **liczba | (liczba)**

Rys. 4.3 Niejednoznaczna gramatyka dla wyrażeń matematycznych z uwzględnieniem operatorów +,-,*,/,()



Rys. 4.4 Dwa drzewa wyprowadzenia dla wyrażenia liczba + liczba * liczba

4.2 Hierarchia gramatyk według Chomsky'ego

Naom Chomsky zdefiniował cztery klasy gramatyk oraz cztery klasy języków formalnych. Klasy te numerowane są od 0 do 3. [2][8][10]

Klasa 0

Gramatykę $G = < N, T, P, S >$, w której produkcje mają postać $\alpha \rightarrow \beta$, gdzie α i β są dowolnymi łańcuchami symboli tej gramatyki, przy czym $\alpha \neq \varepsilon$ nazywamy *semi-gramatykami Thuego*, *gramatykami bez ograniczeń*, *gramatykami struktur frazowych*, *gramatykami kombinatorycznymi* lub *gramatykami klasy 0*.
Definicja gramatyk klasy 0 nie nakłada żadnych ograniczeń na postać produkcji gramatyki w stosunku do ogólnej definicji gramatyki.
Na Rys. 4.5 przedstawiono gramatykę klasy 0 dla napisów postaci a^{2i}.

$$S \rightarrow ACaB$$
$$Ca \rightarrow aaC$$
$$CB \rightarrow DB$$
$$CB \rightarrow E$$
$$Ad \rightarrow Da$$
$$AD \rightarrow AC$$
$$aE \rightarrow Ea$$
$$AE \rightarrow \varepsilon$$

Rys. 4.5 Gramatyka klasy 0 dla napisów $\{a^{2i} \mid i \geq 1\}$

Języki generowane przez gramatyki tego typu noszą nazwę języków *rekurencyjnie przeliczalnych*. Dowodzi się równoważność gramatyk typu 0 i maszyn Turinga. [8]

Klasa 1

Gramatykę $G = <N, T, P, S>$, w której produkcje mają postać $\alpha \rightarrow \beta$, gdzie α i β są takimi łańcuchami symboli tej gramatyki, że łańcuch β jest przynajmniej tak długi jak łańcuch α ($|\alpha| <= |\beta|$) oraz dodatkowo dopuszczona jest produkcja $Z \rightarrow \varepsilon$, jeśli język zawiera słowo puste, nazywamy *gramatykami kontekstowymi*, *gramatykami monotonicznymi*, *gramatykami nieskracalnymi* lub *gramatykami klasy 1*.
Termin kontekstowy pochodzi od tego, że dla każdej gramatyki monotonicznej można znaleźć równoważną jej gramatykę, której produkcje (z wyjątkiem ewentualnej produkcji $Z \rightarrow \varepsilon$) mają postać $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ *gdzie* A jest nieterminalem ($A \in N$), zaś α_1, α_2, β są dowolnymi łańcuchami symboli gramatyki, przy czym $\beta \neq \varepsilon$. Produkcje o tej postaci pozwalają na zastąpienie nieterminala A łańcuchem β tylko w lewostronnym kontekście α_1 i prawostronnym kontekście α_2. Gramatyki kontekstowe mają taką samą siłę wyrazu jak automat liniowo ograniczony. [8]

Klasa 2

Gramatykę G = <N, T, P, S>, w której produkcje mają postać $A \rightarrow x\beta$ lub gdzie A jest nieterminalem ($A \in N$), zaś łańcuch β jest dowolnym łańcuchem symboli tej gramatyki nazywamy gramatykami bezkontekstowymi lub gramatykami klasy 2. Termin bezkontekstowy pochodzi od tego, że produkcje takiej gramatyki pozwalają na bezwarunkowe (bez uwzględniania kontekstu) zastąpienie nieterminala A łańcuchem β. Języki generowane przez gramatyki tego typu noszą nazwę języków bezkontekstowych. Większość języków programowania należy do klasy gramatyk bezkontekstowych. Przykładem gramatyki bezkontekstowej jest gramatyka z Rys. 4.1.[2] Gramatyki bezkontekstowe są równoważne automatom skończonym ze stosem.

Klasa 3

Gramatykę $G = <N, T, P, S>$, w której produkcje mają postać $A \rightarrow x\,\beta$ lub $A \rightarrow x$ gdzie A i B są nieterminalami ($A, B \in N$) zaś łańcuch x jest dowolnym łańcuchem symboli terminalnych tej gramatyki ($x \in T^*$) nazywamy *gramatyką prawostronnie liniową*.
Gramatykę $G = <N, T, P, S>$, w której produkcje mają postać $A \rightarrow \beta\, x$ lub $A \rightarrow x$ gdzie A i B są nieterminalami ($A, B \in N$) zaś łańcuch x jest dowolnym łańcuchem symboli terminalnych tej gramatyki ($x \in T^*$) nazywamy *gramatyką lewostronnie liniową*.

Gramatyki prawostronnie liniowe i lewostronnie liniowe nazywamy gramatykami liniowymi, gramatykami regularnymi lub gramatykami klasy 3.
Na Rys. 4.6 przedstawiono gramatykę prawostronnie liniową dla napisów 0(10)*, natomiast na Rys. 3.1 jest zaprezentowana gramatyka lewostronnie liniowa dla tego samego wzorca.

$$S \rightarrow 0A$$
$$A \rightarrow 10A \mid \varepsilon$$

Rys. 4.6 Prawostronnie liniowa gramatyka dla napisów 0 (10)*

$$S \rightarrow S10 \mid 0$$

Rys. 4.7 Lewostronnie liniowa gramatyka dla napisów 0 (10)*

Tabela 7 Klasy gramatyk, nazwy gramatyk, generowane przez nie języki oraz przykładowe konstrukcje językowe

Klasa	Gramatyka	Język	Przykładowe konstrukcje
0	semi-gramatyka Thuego, bez ograniczeń, struktur frazowych, kombinatoryczna	rekurencyjnie przeliczalny	
1	kontekstowa, monotoniczna, nieskracalna	kontekstowy	$\{a^n b^n c^n \mid n \geq 0\}$
2	bezkontekstowa	bezkontekstowy	struktury zagnieżdżone, poprawnie sparowane nawiasy, wyrażenia
3	prawostronnie liniowa, lewostronnie liniwa	zbiór regularny	dowolne lub konkretne wystąpienia zbioru symboli, identyfikatory

4.3 Analiza zstępująca

Analiza zstępująca (ang. *top-down*) wzięła swoją nazwę od sposobu tworzenia drzewa wyprowadzenia. Drzewo to zaczynamy budować od korzenia, następnie tworzymy potomków korzenia według odpowiednich produkcji gramatycznych i w ten sposób zmierzamy w dół drzewa docierając do liści. Liśćmi są symbole terminalne.

Na analizę zstępującą można patrzeć jak na próbę znalezienia lewostronnego wyprowadzenia dla napisu wejściowego.

Na Rys. 4.8 pokazano lewostronne wyprowadzenie dla napisu *t* = **liczba** + **liczba** * **liczba** według gramatyki z Rys. 4.1.

wyrazenie $\Rightarrow_{lm}$
wyrazenie + *skladnik* $\Rightarrow_{lm}$
skladnik + *skladnik* $\Rightarrow_{lm}$
czynnik + *skladnik* $\Rightarrow_{lm}$
liczba + *skladnik* $\Rightarrow_{lm}$
liczba + *skladnik* * *czynnik* $\Rightarrow_{lm}$
liczba + *czynnik* * *czynnik* $\Rightarrow_{lm}$
liczba + **liczba** * *czynnik* $\Rightarrow_{lm}$
liczba + **liczba** * **liczba**

Rys. 4.8 Lewostronne wyprowadzenie napisu liczba + liczba * liczba

Metoda zejść rekurencyjnych

Metoda zejść rekurencyjnych jest najbardziej popularną i najprostszą metodą parsowania zstępującego. Jest również najmniej efektywna. W tej metodzie, aby przetworzyć dane, wykonuje się grupę rekurencyjnie wywołujących się procedur. Każda z tych procedur jest związana z poszczególnymi nieterminalami z gramatyki. W ogólnym przypadku, każdemu nieterminalowi przyporządkowana jest jedna procedura. Kolejność wywoływania tych procedur, wyznacza niejawnie drzewo wyprowadzenie dla napisu.

Poniżej przedstawiono etapy budowania drzewa składniowego dla gramatyki z rysunku 4.9 i napisu wejściowego *t* = *cad*.

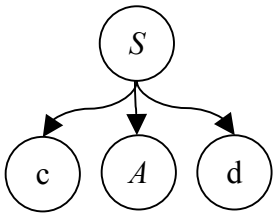
$$S \rightarrow \mathbf{c}A\mathbf{d}$$
$$A \rightarrow \mathbf{ab} \mid \mathbf{a}$$

Rys. 4.9 Gramatyka dla napisu postaci c (ab | b) d

Parsowanie napisu polega na próbie znalezienia jego wyprowadzenia. Podczas analizy, parser może ale nie musi zbudować drzewo wyprowadzenia. Najczęściej jednak robi to, ponieważ hierarchiczna struktura jest wygodna do dalszych etapów kompilacji takich jak sprawdzanie semantyki czy generowanie kodu pośredniego.

Analiza składniowa zaczyna się od wywołania procedury dla symbolu startowego gramatyki. Dla każdego terminala następuje jego porównanie z symbolem bieżącym, natomiast dla nieterminali wywołuje się jego procedurę.

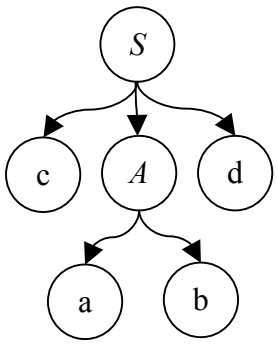
Aby zstępująco zbudować drzewo wyprowadzenia dla napisu *t* = *cad*, na początku tworzy się wierzchołek drzewa i etykietuje się go symbolem startowym gramatyki *S*. Wśród produkcji gramatycznych odnajduje się produkcje, której prawą stroną jest symbol *S* a następnie dla wierzchołka drzewa tworzymy potomków zgodnie z lewą stroną produkcji. Etap ten przedstawiony jest na Rys. 4.10.



Rys. 4.10 Drzewo składniowe dla rozwiniętej produkcji *S* → c*A*d

Można zauważyć, że pierwszy liść po lewej strony oznaczony etykietą *c* pasuje do pierwszego znaku analizowanego napisu. Należy teraz rozpatrzyć wierzchołek drzewa oznaczony *A*. Symbol *A* jest symbolem nieterminalnym, czyli zmienną gramatyczną. Zastępuje on ciąg innych symboli gramatycznych. Żeby się dowiedzieć, jaką produkcję zastępuje symbol *A* trzeba wczytać kolejny symbol napisu *t* i na tej podstawie podjąć decyzje. Kolejnym symbolem po znaku *c* w napisie *t* jest symbol *a*. Ponieważ symbol *A* ma dwie produkcje, i obydwie zaczynają się symbolem terminalnym *a*, nie wiadomo, która produkcja jest właściwa. W tej sytuacji parser wybiera jedną z produkcji, a w przypadku, jeżeli dalsza analiza się nie powiedzie może spróbować inną alternatywę. W omawianym przykładzie, jeżeli parser wybierze produkcję $A \rightarrow \mathbf{ab}$ dalsza analiza się nie powiedzie, dlatego, że chociaż uda się dopasować symbol *a*, parser będzie starał się dopasować znajdujący się w drzewie składniowym symbol *b,* który nie występuje w analizowanym napisie.

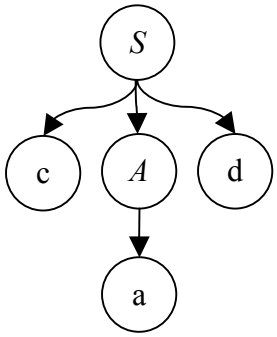
Drzewo składniowe dla tego etapu parskania pokazano na Rys. 4.11.



Rys. 4.11 Drzewo składniowe dla napisu **cabd**

Ponieważ parsowanie nie powiodło się, a parser nie sprawdził jeszcze wszystkich możliwych produkcji, musi wrócić do miejsca wyboru i wypróbować kolejne produkcje. W tym przypadku, nieterminal *A* zostanie rozwinięty według produkcji $A \rightarrow \mathbf{a}$ a drzewo składniowe będzie miało strukturę jak to zaprezentowane na Rys. 4.12.

Warto zaznaczyć, że wracając do miejsca wyboru i przebudowie drzewa, parser musi zwrócić na wejście przeczytane symbole. Zostaną one wczytane jeszcze raz i parser spróbuje dopasować je do nowego drzewa.



Rys. 4.12 Drzewo składniowe dla napisu cad

Program ten nie będzie jednak działał poprawnie. Nie wynika to z błędów programistycznych, ale z tego, że gramatyki z Rys. 4.3 nie można parsować metodami zstępującymi.

Lewostronna rekurencja i metody jej usuwania

Analizator składniowy działający metodą zejść rekurencyjnych może się zapętlić w przypadku produkcji lewostronnie rekurencyjnych. Jeżeli nieterminalowi α odpowiada procedura parsujα, a procedura parsujα na samym początku swojego działania wywołuje siebie rekurencyjnie, nie ma możliwości przerwania tej rekurencji. Parser zapętli się. Rozwiązaniem tego problemu może być przepisanie gramatyki na równoważną w taki sposób, żeby wyeliminować lewostronną rekurencję i zastąpić ją prawostronną rekurencją. Algorytm usuwania lewostronnej rekurencji jest prosty, ale czyni on gramatykę mniej czytelną.

```
void wyrażenie() {
      wyrażenie() ;                        // nieskonczona rekurencja
      if ( biezacy == Leksem::Plus ) wczytaj( Leksem::Plus );
      else if ( biezacy == Leksem::Minus ) wczytaj( Leksem::Minus );
      else throw ParseErrorException;
      skladnik ();
}

void składnik() {
      składnik();                  // nieskonczona rekurencja
      if ( biezacy == Leksem::Star ) wczytaj( Leksem::Star );
      else if ( biezacy == Leksem::Slash ) wczytaj( Leksem::Slash);
      else
            throw ParseErrorException();
      czynnik ();
}

void czynnik() {
      if ( biezacy == Leksem::LeftBranch ) {
            wczytaj( Leksem::LeftBranch )
            wyrażenie ();
      }
      if ( biezacy == Leksem::Id ) wczytaj( Leksem::Id );
}

void wczytaj ( Leksem _leksem ) {
      if ( biezacy == _leksem ) biezacy = daj_nastepny_symbol ( );
      else throw ParseErrorException();
}
```

Rys. 4.13 Pseudokod parsera zstępującego dla gramatyki z Rys. 4.3

Usuwanie lewostronnej rekurencji polega na zamianie produkcji

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$$

na następujące

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Nieterminal A generuje te same napisy ale nie jest już lewostronnie rekurencyjny. Na Rys. 4.14 przedstawiono gramatykę z rysunku z której usunięto lewostronną rekurencję.

wyrazenie → *skladnik wyrazenie'*
wyrazenie' → + *skladnik wyrazenie'*
wyrazenie' → - *skladnik wyrazenie'*
wyrazenie' → ε
skladnik → *czynik skladnik'*
skladnik' → * *czynnik składnik'*
skladnik' → / *czynnik składnik'*
skladnik' → ε
czynnik → **(** *wyrażenie* **)**
czynnik → **liczba**

Rys. 4.14 Gramatyka z Rys. 4.1 po usunięciu lewostronnej rekurencji

RysunekRys. 4.15 przedstawia fragment programu dla gramatyki z Rys. 4.14.

Kod z Rys. 4.15 nie zawiera nieskończonych wywołań rekurencyjnych. Istnieje warunek, w którym rekurencja zostanie przerwana. Warunek ten odpowiada ε-produkcjom z

gramatyki. Na Rys. 4.15 zaprezentowano kod funkcji wyrażenie_() wraz z odpowiadającymi jemu produkcjami gramatycznymi. Produkcje zawierające ε-produkcje po prawej stronie wymagają specjalnego traktowania. Analizator stosujący metodę zejść rekurencyjnych wybiera ε-produkcje wtedy, gdy nie można wybrać żadnej innej produkcji.

```
void wyrażenie() {
      składnik();
      wyrażenie_();
}

void wyrażenie_() {
// wyrazenie_→ + skladnik
      if ( biezacy == Leksem::Plus )
            wczytaj ( Leksem::Plus );

// wyrazenie_→ - skladnik
      else if ( biezacy == Leksem::Minus )
            wczytaj( Leksem::Minus );
else
            return;           // wyrazenie_→ ε
      skladnik ();
      wyrażenie_();
}

void składnik() {
      czynnik();
      składnik_();
}

void składnik_() {
if ( biezacy == Leksem::Star )
            wczytaj( Leksem::Star);
      else if ( biezacy == Leksem::Slash )
            wczytaj( Leksem::Slash);
else
            return;
      czynnik();
      skladnik_();
}

void czynnik() {
      if ( biezacy == Leksem::LeftBranch ) {
            wczytaj( Leksem::LeftBranch )
            wyrażenie ();
      }
      if ( biezacy == Leksem::Id )
            wczytaj( Leksem::Id );
}
```

Rys. 4.15 Pseudokod dla zstępującego parsera dla gramatyki z Rys. 4.14

Faktoryzacja gramatyki

Głównym problemem podczas parsowania metodą zejść rekurencyjnych jest to, że nie zawsze wiadomo, którą z produkcji wybrać do rozwinięcia nieterminala w drzewie składniowym. Niektóre instrukcje języków wysokopoziomowych zaczynają się jednoznacznym słowem kluczowym. W takim przypadku wybranie właściwej produkcji gramatycznej na podstawie jednego wczytanego symbolu jest zadaniem prostym.

```
class nazwaKlasy { cialoKlasy };
if ( wyrazenie ) instrukcja else instrukcja
while ( wyrazenie ) instrukcja
```

Rys. 4.16 Przykłady instrukcji zaczynających się od unikalnych słów kluczowych

Nie wszystkie konstrukcje językowe mogą zaczynać się unikalnym słowem kluczowym. Przykłady takich instrukcji pokazano na Rys. 4.17.

```
double identyfikator { parametryFunkcji ) ;
double identyfikator opcjonalnePrzypisanie ;
```

Rys. 4.17 Przykład dwóch różnych instrukcji zaczynających się tym samym słowem kluczowym

Proces faktoryzacji gramatyki polega na tym, że kiedy nie jest jasne, którą z produkcji wybrać do rozwinięcie nieterminala w drzewie składniowym, należy przepisać tę produkcji, a decyzje o ich wyborze odłożyć do chwili aż przeczytanych zostanie wiecej znaków wejściowych.

Produkcje dla α,

$$\alpha \rightarrow \alpha\beta \mid \alpha\beta\delta$$

po faktoryzacji zostaną zamienione na równoważne następujące

$$\alpha \rightarrow \alpha\beta\phi$$
$$\phi \rightarrow \delta \mid \varepsilon.$$

Przykład zastosowania lewostronnej faktoryzacji dla produkcji deklaracji zmiennej i funkcji z Rys. 4.17 pokazano na Rys. 4.18.

deklaracja → *typ* **identyfikator** *identyfikatorLubFunkcja* ;
identyfikatorLubFunkcja → (*listaParametrow*)
identyfikatorLubFunkcja → *opcjonalnePrzypisanie*

Rys. 4.18 Gramatyka dla instrukcji z Rys. 4.17 po lewostronnej faktoryzacji

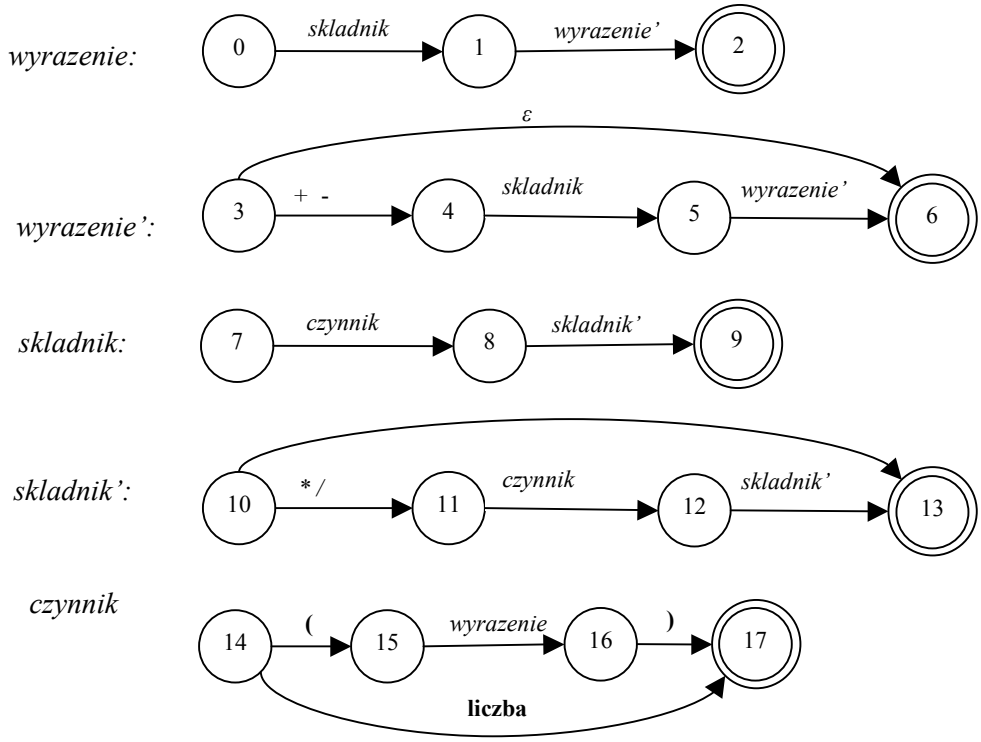
Diagramy przejść dla analizatorów przewidujących

Diagramy przejść w postaci maszyn stanowych są bardzo wygodnym sposobem modelowania zachowania analizatorów leksykalnych. Istnieje również możliwość użycia ich przy budowie parserów.

Korzystając z gramatyki, można zbudować diagram przejść analizatora przewidującego. Należy najpierw usunąć lewostronną rekurencję, a następnie wykonać lewostronną faktoryzację gramatyki. Następnie dla każdego nieterminala należy wykonać następujące kroki.

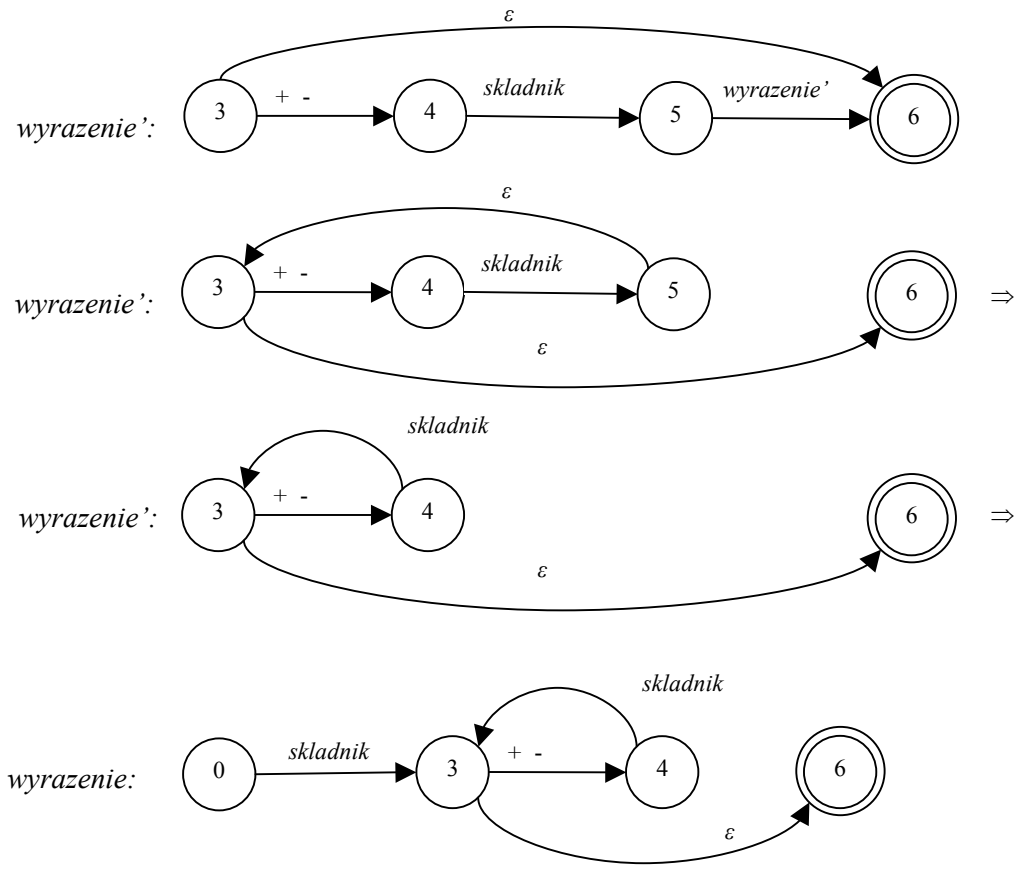
1. stworzyć stany, początkowy i końcowy
2. dla każdej produkcji $A \rightarrow X_1X_2X_3\ldots X_n$ stworzyć ścieżkę od stanu początkowego do końcowego, z krawędziami etykietowanymi X_1, X_2 ,$X_3\ldots X_n$.

Analizator przewidujący rozpoczyna działanie w stanie początkowym dla symbolu startowego. Jeśli po wykonaniu pewnej pracy jest w stanie *s*, z krawędzią etykietowaną *a* prowadzącą do stanu *t* i następnym symbolem wejściowym jest *a*, to analizator przesuwa wskaźnik wejścia i jedną pozycję w prawo i przechodzi do stanu *t*. Jeżeli jednak krawędź jest etykietowana nieterminalem *A*, to analizator przechodzi do stanu startowego dla *A*, nie przesuwając wskaźnika wejścia. Jeśli dojdzie do stanu końcowego *A* z wejścia w czasie przechodzenie do stanu *s* do *t*. Jeżeli krawędź od *s* do *t* z etykietą ε, to analizator przechodzi ze stanu *s* do *t* bez przesuwania wskaźnika wejścia.



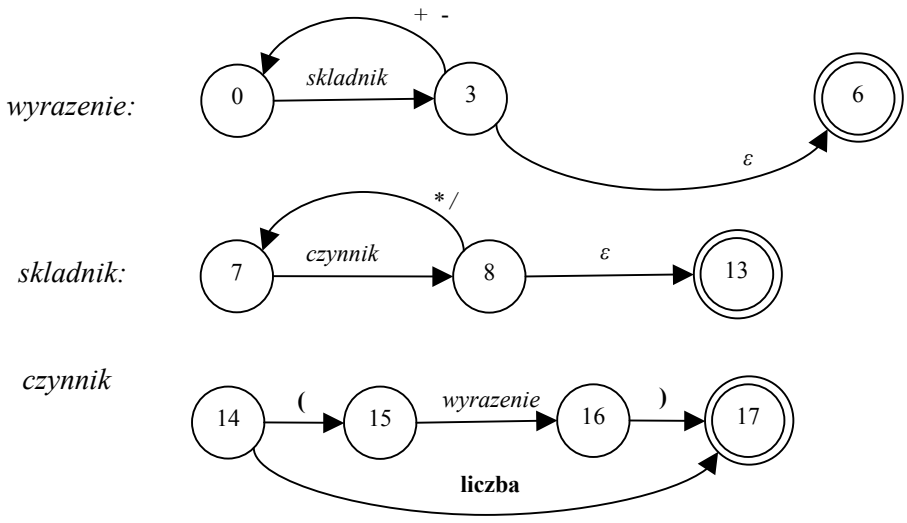
Rys. 4.19 Diagram przejść dla gramatyki z Rys. 4.14

Maszyny stanowe można upraszczać w celu minimalizacji ilości stanów. Warto zaznaczyć, że ilość stanów automatu skończonego ma bezpośredni wpływ na zajętość pamięci przez program działający w oparciu o maszynę stanową.



Rys. 4.20 Uproszczone diagramy przejść z Rys. 4.19

Dokonując przekształceń pokazanych na Rys. 4.20 można otrzymać uproszczoną wersję diagramów z Rys. 4.19. Ta zminimalizowana maszyna stanowa pokazana jest na Rys. 4.21.



Rys. 4.21 Uproszczone diagramy przejść z Rys. 4.20

Analiza przewidująca

W wielu przypadkach, z gramatyki *A* niemożliwej do parsowania matodami zstępującymi, usuwając z niej lewostronną rekurencję oraz wykonując lewostronna faktoryzację, można uzyskać równoważną gramatyką *A*', z której może skorzystać analizator napisany metodą zejść rekurencyjnych niepotrzebujący nawrotów.
W ogólnym przypadku, znalezienie odpowiedniej produkcji dla nieterminala może wymagać zastosowania metody prób i błędów. To znaczy, możemy próbować zastosować jakąś produkcję i gdy to się nie uda, wrócić do miejsca wyboru i zacząć próbować inne produkcje. Jeśli, po zastosowaniu produkcji, nie uda się dopasować drzewa do napisu wejściowego, to wybrana produkcja nie jest właściwa. Istnieje jednak specjalny przypadek, zwany przewidującą analizą składniową, w którym nie ma nawrotów i prób zastosowania różnych produkcji.
Podczas budowania drzewa, dla każdego nieterminala produkcje gramatyki są wybierane na podstawie symbolu bieżącego. Jeżeli prawa strona produkcji zaczyna się od symbolu terminalnego, wybranie właściwej produkcji jest proste. W przypadku, gdy kilka produkcji zaczyna się tym samym symbolem terminalnym istnieje potrzeba sprawdzenia wszystkich tych produkcji. Jeżeli prawa strona produkcji zaczyna się od nieterminala wybór właściwej produkcji jest o wiele trudniejszy. Przykład takich produkcji pokazano na Rys. 4.22.

instrukcja → *instrukcjaWyboruIf*
instrukcja → *instrukcjaWyboruSwitch*
instrukcja → *instrukcjaIteracjiFor*
instrukcja → *instrukcjaIteracjiWhile*
instrukcja → *instrukcjaIteracjiDoWhile*

Rys. 4.22 Przykład produkcji dla instrukcji języka C++

W przypadku, gdy produkcja zaczyna się nieterminalem, produkcję tę można wybrać, jeżeli symbol bieżący może zostać wygenerowany z tego nieterminala. Przewidująca analiza składniowa opiera swoje działanie na informacji, które symbole mogą być wygenerowane jako pierwsze przez prawe strony produkcji. Dla produkcji $A \rightarrow \alpha$, można zdefiniować zbiór FIRST(α) jako zbiór wszystkich symboli leksykalnych, które mogą zaczynać napisy wygenerowane z α. Przykład zbiorów FIRST dla produkcji z gramatyki na Rys. 4.22 przedstawiono na Rys. 4.23.

FIRST(*instrukcjaWyboruIf*) = { **if** }
FIRST(*instrukcjaWyboruSwitch*) = { **switch** }
FIRST(*instrukcjaIteracjiFor*) = { **for** }
FIRST(*instrukcjaIteracjiWhile*) = { **while** }
FIRST(*instrukcjaIteracjiDoWhile*) = { **do** }
FIRST(*wyrazenie*) = { **liczba**, **(** }

Rys. 4.23 Zbiory FIRST dla instrukcji z Rys. 4.22

Zbiory First i Follow

Zbiory FIRST muszą być rozpatrywane wtedy, gdy istnieją dwie produkcje $A \rightarrow \alpha$ i $A \rightarrow \beta$. Aby można było stosować metodę zejść rekurencyjnych bez nawrotów, zbiory FIRST(α) i FIRST(β) muszą być rozłączne. Dopiero wtedy bieżący symbol umożliwia jednoznaczne podjęcie decyzji, którą produkcję wybrać. Jeśli symbol bieżący należy do FIRST(α) wybieramy produkcję α, jeżeli natomiast bieżący symbol należy do FIRST(β) wybieramy produkcję β.

Aby obliczyć FIRST(X) dla wszystkich symboli z gramatyki X, należy działać zgodnie z poniższymi regułami, aż do żadnego ze zbiorów FIRST nie będzie już można dodać żadnych terminali ani ε.

1. Jeśli X jest terminalem, to FIRST(X) jest równe $\{X\}$
2. Jeśli $X \rightarrow \varepsilon$ jest produkcją, to należy dodać ε do FIRST(X)
3. Jeśli X jest nieterminalem i $X \rightarrow Y_1 Y_2 \ldots Y_k$ jest produkcją, to a trzeba umieścić w FIRST(X), jeżeli istnieje takie i, że a jest w FIRST(Y_i), a ε jest we wszystkich FIRST(Y_1),…,FIRST(Y_{i-1}), to znaczy gdy $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$. Jeśli ε jest w FIRST(Y_j) dla wszystkich j = 1, 2, …, k, to do FIRST(X) należy dodać ε.

Przykładowo, wszystkie symbole z FIRST(Y_1) są w FIRST(X). Jeśli z Y_1 nie da się wyprowadzić ε, to do FIRST(X) nie dodajemy nic więcej, ale jeżeli $Y_1 \Rightarrow^* \varepsilon$, to dodajemy FIRST($Y_2$) i tak dalej.

Niech G będzie gramatyką, taką, że $G = < N, T, P, S >$ i niech G należy do gramatyk bezkontekstowych. Formalną definicją zbioru FIRST_k jest
$\text{FIRST}_k(\alpha) = \{ x \in T^* \mid (\alpha \Rightarrow^* x\beta \wedge |x| = k) \vee (\alpha \Rightarrow^* x \wedge |x| = k)\}$ gdzie $\alpha, \beta \in (N \cup T)^*$

Zbiór $\text{FIRST}_k(\alpha)$ jest zbiorem wszystkich terminalnych przedrostków długości k (lub mniejszej, jeżeli z α wyprowadza się łańcuch terminalny krótszy niż k) łańcuchów, które mogą być wyprowadzone z α. Dla wygody notacyjnej FIRST_1 oznacza się jako FIRST.

Dla produkcji $A \rightarrow \alpha$, można zdefiniować również zbiór FOLLOW(α) jako zbiór terminali a, które mogą wystąpić bezpośrednio na prawo od A w pewnej formie zdaniowej, czyli jako zbiór terminali a, takich, dla których istnieje wyprowadzenie o postaci $S \Rightarrow^* \alpha A a \beta$ dla jakichś α i β.

Aby obliczyć FOLLOW(A) dla nieterminali A, należy stosować poniższe reguły, aż do żadnego ze zbiorów FOLLOW nie będzie można już nic dodać.

1. W FOLLOW(S), gdzie S jest symbolem startowym, trzeba umieścić \$, znacznik prawego końca wejścia.
2. Jeśli mamy produkcję $A \rightarrow \alpha B \beta$, to wszystkie symbole z FIRST(β), z wyjątkiem ε, należy umieścić w FOLLOW(B).
3. Jeśli mamy produkcję $A \rightarrow \alpha$B albo produkcję $A \rightarrow \alpha B \beta$, gdzie FIRST($\beta$) zawiera ε (tj. $\beta \Rightarrow^* \varepsilon$), to wszystkie symbole z FOLLOW(A) są w FOLLOW(B).

Niech G będzie gramatyką, taką, że $G = < N, T, P, S >$ i niech G należy do gramatyk bezkontekstowych. Formalną definicją zbioru FOLLOW_k jest

$\text{FOLLOW}_k(\beta) = \{ x \in T^* \mid (Z \Rightarrow^* \alpha\beta\delta \wedge x \in \text{FIRST}_k(\delta))\}$ gdzie $\alpha, \beta, \delta \in (N \cup T)^*\}$

Zbiór $\text{FOLLOW}_k(\beta)$ zawiera terminalne łańcuchy o długości k (lub mniejszej, jeżeli z β wyprowadza się łańcuch terminalny krótszy niż k) które mogą pojawić się w wyprowadzeniach jako następniki β. Dla wygody notacyjnej FOLLOW_1 oznacza się jako FOLLOW.

Na Rys. 4.24 przedstawiono zbiory FIRST i FOLLOW dla nieterminali z gramatyki z Rys. 4.14.

FIRST(*wyrazenie*) = FIRST(*skladnik*) = FIRST(*czynnik*) = { **(**, **liczba** }
FIRST(*wyrazenie'*) → { +, – , ε }
FIRST(*skladnik'*) → { *, /, ε }
FOLLOW(*wyrazenie*) = FOLLOW(*wyrazenie'*) = { **)**, $ }
FOLLOW(*skladnik*) = FOLLOW(*skladnik'*) = { +, –, **)**, $ }
FOLLOW(*czynnik*) = { +, –, *, /, $ }

Rys. 4.24 Zbiory FIRST i FOLLOW dla symboli z gramatyki Rys. 4.14

Budowa przewidującego analizatora składniowego

Przewidujący analizator składniowy jest programem zawierającym funkcję dla każdego nieterminala. Każda funkcja wykonuje czynności:

1) Na podstawie symbolu bieżącego musi zadecydować o wyborze produkcji, Produkcja z prawą stroną α jest wybierana, jeśli symbol bieżący należy do FIRST(α). Jeśli istnieje symbol należący do zbiorów FIRST dla co najmniej dwóch różnych produkcji dla tego samego nieterminala, to metoda przewidująca nie może być zastosowana. ε-produkcja jest wybierana, gdy symbol bieżący nie należy do żadnego ze zbiorów FIRST dla pozostałych produkcji.
2) Ciało funkcji odpowiada prawej stronie wybranej produkcji. Dla nieterminala w kodzie znajduje się wywołanie procedury dla tego nieterminala a dla symbolu terminalnego sprawdza się czy taki sam jak bieżący symbol. Jeżeli w jakimś miejscu symbol bieżący jest różny od symbolu terminalnego, zgłaszany jest błąd.

Przewidujący analizator składniowy oparty na diagramach przejść

Jeżeli diagram przejść dla analizatora, będzie diagramem deterministycznym utworzony na jego podstawie parser będzie mógł działać metodą przewidującą.

Tablice analizatorów przewidujących

Analizator przewidujący sterowany tablicami składa się z wejścia, stosu, tablicy analizatora składniowego i wyjścia. Na wejściu są symbole analizowanego napisu zakończone symbolem $. Stos służy do przetrzymywania symboli gramatyki, a na samym jego dole znajduje się znak $. Na samym początku analizy na stosie znajduje się symbol startowy gramatyki, położony na znak $. Tablica analizatora jest dwuwymiarową tablicą M[*A*, *a*], gdzie *A* jest nieterminalem, a *a* jest terminalem bądź symbolem prawego końca napisu wejściowego $.

Parser podczas analizy bieże pod uwagę *X*, symbol na wierzchołku stosu, oraz aktualny symbol na wejściu *a*.

1) Jeśli $X = a = \$$, to analizator pomyślnie kończy analizę napisu.
2) Jeśli $X = a \neq \$$, to analizator zdejmuje *X* ze stosu i wczytuje następny symbol
3) Jeśli *X* jest nieterminalem, to program sprawdza wartość *M*[*X*, *a*] w tablicy analizatora *M*. Tą wartością będzie albo *X*-produkcja gramatyki, albo wartość **błąd**. Jeżeli wartością jest produkcja gramatyczna, parser zdejmuje *X* z wierzchołka stosu, a na wierzchołek stosu wkłada symbole z prawej stron produkcji. Jeżeli wartością jest **błąd**, parser może obsłużyć błąd lub przerwać analizę.

W celu zbudowania tablicy analizatora M, możemy posłużyć się następującym algorytmem.

1) Dla każdej produkcji $A \rightarrow \alpha$ z gramatyki, wykonaj kroki 2 i 3
2) Dla każdego terminala a z FIRST(α) dodaj $A \rightarrow \alpha$ do M[A, a]
3) Jeśli ε jest w FIRST(α), dodaj $A \rightarrow \alpha$ do M[A, b] dla każdego terminala b z FOLLOW(A). Jeśli ε jest w FIRST(α) oraz \$ jest w FOLLOW(A), dodaj $A \rightarrow \alpha$ do M[A, \$].
4) W pozostałe puste pozycje tablicy M wpisz wartość **błąd**.

Zasada działania analizatora opartego na tablicy jest następująca. Niech $A \rightarrow \alpha$ jest produkcją i a jest w FIRST(α). Jeżeli aktualnym symbolem wejściowym jest a, wtedy analizator rozwija A, używając α. Problem pojawia się tylko wtedy, gdy $\alpha = \varepsilon$ lub $\alpha \Rightarrow^* \varepsilon$. W takim przypadku, jeśli aktualny symbol wejściowy jest w FOLLOW(A) musimy jeszcze raz rozwinąć A używając α.

Algorytm nierekurencyjnej analizy przewidującej

Wejściem dla algorytmu jest napis w oraz tablica analizatora M dla gramatyki G.
Wyjściem jest lewostronne wyprowadzenie dla w jeśli w jest w $L(G)$, lub informacja o błędzie w przeciwnym razie.
Początkowo analizator jest w konfiguracji, w której na stosie jest #S, gdzie S jest startowym symbolem gramatyki. W buforze wejściowym jest w\$. Program, który korzysta z tablicy analizatora przewidującego M do zbudowania wyprowadzenia dla wejścia pokazany jest na Rys. 4.25.

```
niech ip wskazuje pierwszy symbol w$;
do{
      niech X będzie symbolem z wierzchołka stosu,
      a symbolem wskazywanym przez ip;
      if X jest terminalem albo $
            if X = a
                  zdejmij X ze stosu i przesuń ip w przód;
            else
                  return Błąd;
      else {// X jest terminalem
            if M[X, a] = X → Y1Y2 … Yk{
                  zdejmij X ze stosu;
                  połóż Yk, Yk-1, … , Y1 na stosie, z Y1 na wierzchołku;
                  wypisz produkcję X → Y1 Y2 … Yk
            }
            else
                  return Błąd;
      }
} while ( X = $ ); // stos jest pusty
return ZakończonoPomyślnieAnalizę;
```

Rys. 4.25 Pseudokod parsera działającego metodą zstępującą [2]

W Tabela 8 zaprezentowano tablicę analizatora M dla gramatyki z Rys. 4.14. Każdemu wierszowi odpowiada nieterminal n, natomiast kolumnie symbol terminalny t. Na przecięciu wiersza i kolumny znajduj się produkcja, która zostanie użyta do rozwinięcia nieterminalu n w drzewie składniowym. Jeżeli przecięcie wiersza i kolumny jest wartością pustą, oznacza to, że nie można rozwinąć nieterminala n jeśli bieżącym symbolem jest terminal t i oznacza to błąd parsowania.

Tabela 8 Tablica przejść dla gramatyki z Rys. 4.14 [2]

NIETERMINAL	SYMBOL WEJŚCIOWY							
	liczba	**(**	**)**	**+**	**-**	*****	**/**	**$**
wyr	*wyr* → *skl wyr'*	*wyr* → *skl wyr'*						
wyr'			*wyr'* → ε	*wyr'* → *+skl wyr'*	*wyr'* → *-skl wyr'*			*wyr'* → ε
skl	*skl* → *czyn skl'*	*skl* → *czyn skl'*						
skl'			*skl'* → ε			*skl'* → ** czyn skl'*	*skl'* → */ czyn skl'*	
czyn	*czyn* → **liczba**	*czyn* → *(wyr)*						

W Tabela 9 zaprezentowano zawartość stosu, stan wejścia oraz akcje parsera podczas analizy napisu **liczba + liczba * liczba**. Napis jest poprawnie sparsowany, jeżeli na koniec analizy na stosie znajduje się tylko symbol # a na wejściu symbol $.

Tabela 9 Stan stosu, symbole na wejściu oraz wykonywane akcje podczas parsowania napisu **liczba + liczba * liczba [2]**

STOS	WEJŚCIE	WYJŚCIE
# *wyrazenie*	**liczba + liczba * liczba $**	
# *wyrazenie' skladnik*	**liczba + liczba * liczba $**	*wyrazenie* → *skladnik wyrazenie'*
# *wyrazenie' skladnik' czynnik*	**liczba + liczba * liczba $**	*skladnik* → *czynnik skladnik'*
# *wyrazenie' skladnik'* **liczba**	**liczba + liczba * liczba $**	*czynnik* → **liczba**
# *wyrazenie' skladnik'*	**+ liczba * liczba $**	
# *wyrazenie'*	**+ liczba * liczba $**	*skladnik'* → ε
# *wyrazenie' skladnik +*	**+ liczba * liczba $**	*wyrazenie'* → *+ składnik wyrazenie'*
# *wyrazenie' skladnik*	**liczba * liczba $**	
# *wyrazenie' skladnik' czynnik*	**liczba * liczba $**	*skladnik* → *czynnik skladnik'*
# *wyrazenie' skladnik'* **liczba**	**liczba * liczba $**	*czynnik* → **liczba**
# *wyrażenie' skladnik'*	*** liczba $**	
# w*yrazenie' skladnik' c*zynnik*****	*** liczba $**	*skladnik'* → ** czynnik skladnik'*
# *wyrazenie' skladnik' czynnik*	**liczba $**	
# *wyrazenie' skladnik'* **liczba**	**liczba $**	*czynnik* → **liczba**
# *wyrazenie' skladnik'*	**$**	
# *wyrazenie'*	**$**	*skladnik'* → ε
#	**$**	*wyrazenie'* → ε

Gramatki LL(1)

Dla gramatyk z lewostronną rekurencją lub niejednoznacznych, w tablicy M będzie co najmniej jedna pozycja z wieloma wartościami.

Przykład takiej niejednoznacznej gramatyki zaprezentowano na Rys. 4.26 a tablicę przejść dla niej zaprezentowano w Tabela 10.

instrukcja → **if** *wyrazenie instrukcja instrukcja'* | *jakasInstrukcja*
instrukcja' → **else** *instrukcja* | ε
wyrazenie → *jakiesWyrazenie*

Rys. 4.26 Gramatyka instrukcji '*else if*'

Na pozycji M[*instrukcja'*, **else**] są jednocześnie dwie produkcje. Ta gramatyka jest niejednoznaczna. Niejednoznaczność można zauważyć przy wyborze produkcji do zastosowania, gdy na wejściu jest terminal **else**. Istnieje wtedy problem do której klauzuli if **odnosi się ten** else**.**

Tabela 10 Tablica przejść dla instrukcji *'else if' [2]*

NIETERMINAL	SYMBOL WEJŚCIOWY				
	jakasInstrukcja	*Jakies Wyrazenie*	**if**	**else**	**$**
instrukcja	*instrukcja* → jakasInstrukcja		*instrukcja* → **if** *wyrazenie instrukcja instrukcja'*		
instrukcja'				*instrukcja'* → *else instrukcja* *instrukcja'* → ε	instrukcja' → ε
wyrazenie		*wyrazenie* → *jakiesWyrazenie*			

Gramatyka, dla której tablica analizatora nie ma pozycji z wieloma wartościami, jest nazywana gramatyką LL(1). Pierwsze „L" w LL(1) oznacza przeglądanie wejścia od lewej do prawej. Drugie „L" oznacza, lewostronne wyprowadzenie (ang. *leftmost*), „1" oznacza, że do podejmowania decyzji o wyborze produkcji można wczytać co najwyżej jeden symbol terminalny.

Niech dana będzie gramatyka *G* taka, że:

$$G \rightarrow \alpha \mid \beta$$

Można wykazać, że gramatyka *G* jest gramatyką LL(1) wtedy i tylko wtedy, gdy:
1. Dla każdego terminala *a*, z α i β nie daje się jednoznacznie wyprowadzić ciągu rozpoczynającego się od α.
2. Co najwyżej dla jednego z α i β daje się wyprowadzić pusty ciąg.
3. Jeśli $\beta \Rightarrow^* \varepsilon$, to z α nie można wyprowadzić żadnego ciągu rozpoczynającego się od terminala z FOLLOW(*G*).

Dla programu

```
if ( wyrazenie )
      if ( wyrazenie )
            jakasInstrukcja
      else
            jakasInstrukcja
```

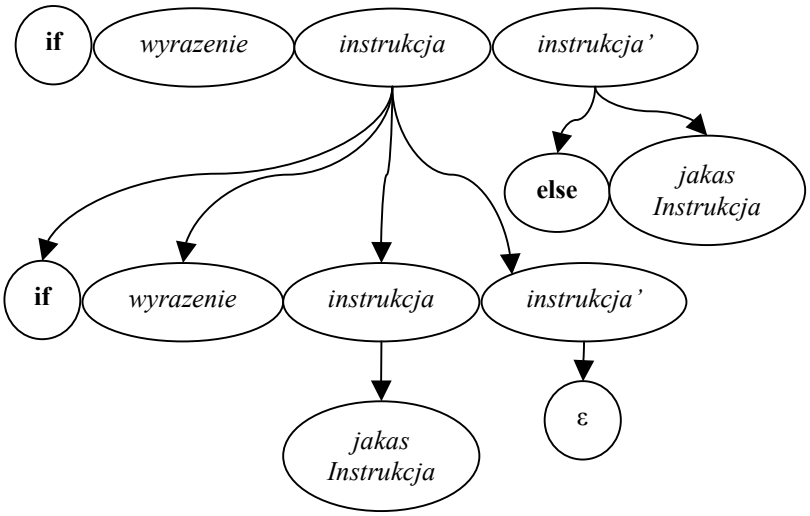
poprawną ewaluacją jest oczywiście drzewo składniowe pokazane na Rys. 4.28. Drzewo będzie budowane w taki sposób, jeżeli parser będzie wybierał produkcję

instrukcja' → *else instrukcja*

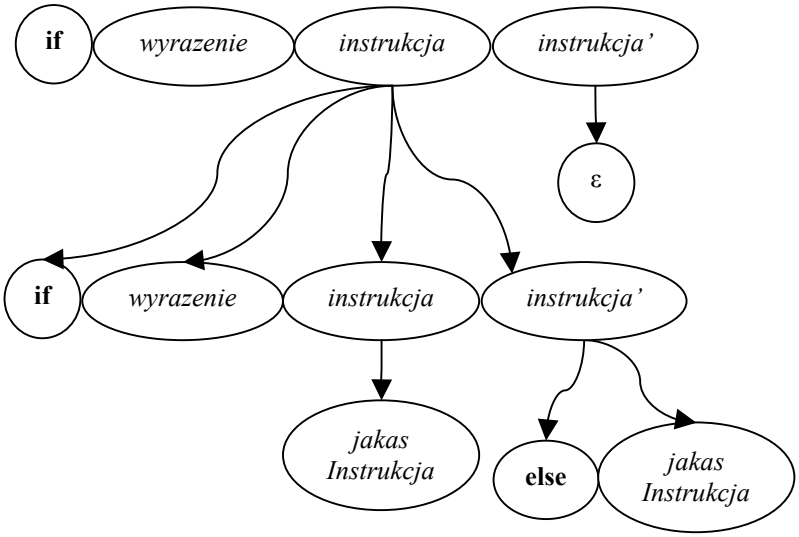
zamiast

instrukcja' → ε.

Taki wybór odpowiada przyłączaniu **else** do najbliższego warunku **if.**



Rys. 4.27 Drzewo składniowe dla instrukcji '*if else'*, gdzie **else** jest skojarzone z 'zewnętrznym' **if**



Rys. 4.28 Drzewo składniowe dla instrukcji '*if else*', gdzie **else** jest skojarzone z najbliższym **if**

4.4 Analiza wstępująca

Analiza wstępująca, zwana również redukującą, buduje drzewo składniowe zaczynając od liści, łącząc je w coraz większe i większe konstrukcje gramatyczne aż do korzenia, który jest symbolem startowym gramatyki. Analiza ta jest równoważna redukcji napisu wejściowego do symbolu startowego gramatyki. W każdym kroku redukcji, pewien podciąg pasujący do prawej strony produkcji jest zastępowany symbolem z lewej strony tej produkcji, i jeśli podciąg jest poprawnie wybierany w każdym kroku, to śledzimy odwrotność prawostronnego wyprowadzenia.

Uchwyt

Uchwytem prawostronnej formy zdaniowej γ nazywamy produkcje $A \rightarrow \beta$ i pozycję w γ, na której znajduje się ciąg symboli β, który należy zastąpić przez A, aby otrzymać poprzednią prawostronną formę zdaniową w prawostronnym wyprowadzeniu γ.

Niech $S \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm} \alpha\beta w$, to $A \rightarrow \beta$ na pozycji po α jest uchwytem $\alpha\beta w$. Ciąg symboli po prawej stronie uchwytu zawiera tylko symbole terminalne. Jeżeli gramatyka jest jednoznaczna, to każda prawostronna forma zdaniowa z tej gramatyki ma dokładnie jeden uchwyt. Dla gramatyk niejednoznacznych może istnieć więcej uchwytów.

Na Rys. 4.29 przedstawiono prawostronne wyprowadzenie napisu *t* = **liczba** + **liczba** * **liczba** według gramatyki z Rys. 4.1.

wyrazenie $\Rightarrow_{rm}$
wyrazenie + *skladnik* $\Rightarrow_{rm}$
wyrazenie + *skladnik* * *czynnik* $\Rightarrow_{rm}$
wyrazenie + *skladnik* + **liczba** $\Rightarrow_{rm}$
wyrazenie + *czynnik* * **liczba** $\Rightarrow_{rm}$
wyrazenie + **liczba** * **liczba** $\Rightarrow_{rm}$
skladnik + **liczba** * **liczba** $\Rightarrow_{rm}$
czynnik + **liczba** * **liczba** $\Rightarrow_{rm}$
liczba + **liczba** * **liczba**

Rys. 4.29 Prawostronne wyprowadzenie napisu **liczba + liczba * liczba**

Tabela 11 zawiera prawostronne formy zdaniowe ich uchwyty oraz produkcje zastosowane do redukcji dla wyprowadzenia z Rys. 4.32. Dla lepszej czytelności terminale oznaczono indeksami 1, 2, 3.

Niech w będzie zdaniem z gramatyki G, w = γn, gdzie γn jest *n*-tą prawostronną formą zdaniową w prawostronnym wyprowadzeniu.

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$$

Aby odtworzyć to wyprowadzenie od końca, wyszukuje się uchwytu β_n w γ_n i zastępuje się β_n lewą stroną pewnej produkcji $A_n \rightarrow \beta_n$ otrzymując (n-1)-szą prawostronną formę zdaniową y_{n-1}. Proces ten powtarza się, wyszukując uchwyt β_{n-1} w γ_{n-1} i redukując ten uchwyt tak, aby otrzymać prawostronną formę γ_{n-2}. Jeżeli uda się dojść do formy zdaniowej γ_0 i zredukować ją do symbolu startowego gramatyki *S* analiza zakończona jest powodzeniem.

Tabela 11 Formy zdaniowe, ich uchwyty oraz używane produkcje podczas parsowania napisu **liczba + liczba * liczba**

PRAWOSTRONNA FORMA ZDANIOWA	UCHWYT	PRODUKCJA UŻYWANA W REDUKCJI
liczba$_1$ + **liczba**$_2$ * **liczba**$_3$	**liczba**$_1$	*czynnik* → **liczba**
czynnik + **liczba**$_2$ * **liczba**$_3$	*czynnik*	*składnik* → *czynnik*
składnik + **liczba**$_2$ * **liczba**$_3$	*składnik*	*wyrazenie* → *składnik*
wyrazenie + **liczba**$_2$ * **liczba**$_3$	**liczba**$_2$	*czynnik* → **liczba**
wyrazenie + *czynnik* * **liczba**$_3$	*czynnik*	*składnik* → *czynnik*
wyrazenie + *składnik* * **liczba**$_3$	**liczba**$_3$	*czynnik* → **liczba**
wyrazenie + *składnik* * *czynnik*	*składnik* * *czynnik*	*składnik* → *składnik* * *czynnik*
wyrazenie + *składnik*	*wyrazenie* + *składnik*	*wyrazenie* *wyrazenie* + *składnik*
wyrazenie		

Stos i jego zastosowanie w analizie redukującej

Efektywną metodą implementacji analizatora redukującego jest użycie stosu do pamiętania symboli gramatyki które do tej pory pojawiły się na wejściu. Niech *$* oznacza prawy koniec wejścia a # dno stosu parsera. Początkowo stos jest pusty, a na wejściu jest napis *t*. Analizator przesuwa zero lub więcej symboli z wejścia na stos, aż na wierzchołku stosu będzie uchwyt *B*. Uchwyt *B* jest wtedy redukowany do lewej strony odpowiedniej produkcji. Parser powtarza ten cykl aż do wystąpienia błędu albo do czasu, gdy na stosie będzie symbol startowy, a wejście będzie puste.
Warto zauważyć, że uchwyt zawsze znajduje się na wierzchołku stosu.
Prefiksem prawostronnej formy zdaniowej, która może wystąpić na stosie analizatora redukującego, nazywany jest *prefiksem żywotnym*.
Tabela 12 przedstawia stan stosu, wejście parsera oraz operacje wykonywane przez parser podczas analizy napisu *t* = **liczba** + **liczba** * **liczba**.

Tabela 12 Stan stosu, symbole na wejściu oraz akcje podczas parsowania napisu **liczba + liczba * liczba**

STOS	WEJŚCIE	AKCJA
#	**liczba**$_1$ + **liczba**$_2$ * **liczba**$_3$ $	przesunięcie
# **liczba**$_1$	+ **liczba**$_2$ * **liczba**$_3$ $	R *czynnik* → **liczba**
# *czynnik*	+ **liczba**$_2$ * **liczba**$_3$ $	R *składnik* → *czynnik*
# *składnik*	+ **liczba**$_2$ * **liczba**$_3$ $	R *wyrazenie* → *składnik*
# *wyrazenie*	+ **liczba**$_2$ * **liczba**$_3$ $	przesunięcie
# *wyrazenie* +	**liczba**$_2$ * **liczba**$_3$ $	przesunięcie
# *wyrazenie* + **liczba**$_2$	* **liczba**$_3$ $	R *czynnik* → **liczba**
# *wyrazenie* + *czynnik*	* **liczba**$_3$ $	R *składnik* → *czynnik*
# *wyrazenie* + *składnik*	* **liczba**$_3$ $	przesunięcie
# *wyrazenie* + *składnik* *	**liczba**$_3$ $	przesunięcie
# *wyrazenie* + *składnik* * **liczba**$_3$	$	R *czynnik* → **liczba**
# *wyrazenie* + *składnik* * czynnik	$	R *składnik* → *składnik* * *czynnik*
# *wyrazenie* + *składnik*	$	R *wyrazenie* → *wyrazenie* + *składnik*
# *wyrazenie*	$	akceptowanie wejścia

Istnieją cztery akcje, które może wykonać parser. Przedstawiono je wraz z opisem działania w poniższej tabeli.

Tabela 13 Akcje możliwe do wykonania przez parser

AKCJA	OPIS DZIAŁANIA
przesunięcie	Wstawienie kolejnego symbolu z wejścia na wierzchołek stosu.
redukcja	Na wierzchołku stosu znajduje się prawy koniec uchwytu. Parser musi znaleźć lewy koniec uchwytu i zdecydować, według jakiej produkcji zastąpić uchwyt. Nieterminal który powstaje po zredukowaniu uchwytu jest kładziony na wierzchołek stosu.
akceptowanie	Parser wczytał cały napis wejściowy i dokonał jego redukcji do symbolu startowego gramatyki, który znajduje się na wierzchołku stosu parsera.
błąd	Parser nie może dokonać pełnej analizy napisu. Napis nie pasuje do gramatyki.

Błędy podczas analizy redukującej

Istnieją gramatyki bezkontekstowe, których nie można analizować metodami wstępującymi. Dla takich gramatyk, parser, pomimo znajomości zawartości stosu i następnego symbolu nie może zdecydować, jaką akcję wykonać. Może się wahać pomiędzy przesunięciem lub redukcją, oraz pomiędzy dwoma redukcjami.

Przykład niejednoznacznej gramatyki dla „*wiszącego else*” pokazano na Rys. 4.30.

instrukcja → **if** *wyrazenie instrukcja*
instrukcja → **if** *wyrazenie instrukcja* **else** *instrukcja*

Rys. 4.30 Niejednoznaczna gramatyka dla instrukcji warunkowej

Jeżeli na stosie stos parsera są symbole,

if *wyrazenie instrukcja*

a następnym symbolem zwróconym przez lekser jest **else**, parser nie może zdecydować, czy powinien wykonać redukcję według produkcji

instrukcja → **if** *wyrazenie instrukcja*,

czy powinien wczytać **else**, odłożyć go na stos i kontynuować analizę starając się dopasować jak najdłuższy napis. Błąd ten jest nazywany błędem typu przesunięcie – redukcji (ang. *shift – recudtion conflict*). Rozwiązaniem tego konfliktu w kompilatorach, jest ustawienie *przesunięcia* jako domyślnej akcji.

Innym częstym błędem jest błąd redukcji – redukcji (ang. *reduction – reduction conflict*). Jest to sytuacja, w której parser pomimo znajomości zawartości stosu i kolejnego symbolu na wejściu nie potrafi wybrać, według której produkcji należy zredukować uchwyt. Sytuacja ta ma miejsce najczęściej, jeżeli w gramatyce istnieją dwie produkcje o takich samych prawych stronach.

Przykład takiej niejednoznacznej gramatyki pokazano na Rys. 4.31.

wywolanieFunkcji → **identyfikator** (*listaArgumentow*)
listaArgumentow → *listaArgumentow* , *argument*
listaArgumentow → *argument*
argument → **identyfikator**
wyrazenie → **identyfikator** (*listaWyrazen*)
wyrazenie → **identyfikator**
listaWyrazen → *listaWyrazen* , *wyrazenie*
listaWyrazen → *wyrazenie*

Rys. 4.31 Niejednoznaczna gramatyka dla wywołań funkcji i indeksowania zmiennych tablicowych

Niejednoznaczność gramatyki, wynika z tego, że instrukcja A(B, C), może oznaczać wywołanie funkcji jak również odwołanie do zmiennej tablicowej z użyciem operatora indeksowania.

Niech na stosie parsera będą następujące symbole,

… **identyfikator** (**identyfikator**.

Terminal **identyfikator** na wierzchołku stosu jest uchwytem, jednak parser nie potrafi wybrać produkcji, według której zredukować ten uchwyt. Jeżeli A jest procedurą, parser powinien użyć produkcji

argument → **identyfikator**

natomiast, jeżeli A jest zmienną tablicową właściwym wyborem parsera powinno być

wyrazenie → **identyfikator**.

Rozwiązaniem tego problemu może być, pozwolenie, aby parametrem aktualnym funkcji było każde wyrażenie. Poprawioną gramatyką pokazano na Rys. 4.32.

wywolanieFunkcji → **identyfikator** (*listaArgumentow*)
listaArgumentow → *listaArgumentow* , *argument*
listaArgumentow → *argument*
argument → *wyrazenie*
wyrazenie → **identyfikator** (*listaWyrazen*)
listaWyrazen → *listaWyrazen* , *wyrazenie*
listaWyrazen → *wyrazenie*
wyrazenie → *skladnik* → *czynnik* → **identyfikator**

Rys. 4.32 Jednoznaczna gramatyka dla wywołań funkcji i indeksowania zmiennych tablicowych

Analiza LR

Analiza LR(*k*) jest bardzo wydajną metodą parsowania. Jej pożądaną właściwością jest to, że można nią parsować prawie wszystkie programistyczne konstrukcje językowe dające opisać się gramatyką bezkontekstową.

Pomiędzy gramatykami LL i LR jest zasadnicza różnica. Aby gramatyka była LR(*k*), musi móc rozpoznać wystąpienia prawej strony produkcji po zobaczeniu wszystkiego, co z tej produkcji zostało wyprowadzone i po obejrzeniu *k* symboli z wejścia. Jest to dużo słabsze ograniczenia niż to narzucane gramatykom LL(*k*). W gramatykach LL(*k*) trzeba rozpoznać użycie produkcji zobaczywszy tylko *k* pierwszych symboli wyprowadzonych z jej prawej

strony. Zbiór języków generowanych przez gramatykę LL jest podzbiorem języków generowanych przez gramatykę LR.

$$L(\text{ LL}(k)\text{ }) \subset L(\text{ LR}(k)\text{ })$$

„L" oznacza, że analizujemy napis od lewej do prawej strony, „R" (ang. *rightmost*), że staramy się odnaleźć prawostronne drzewo wyprowadzenia, *k* ilość symboli potrzebnych do podjęcia decyzji o następnej akcji, jaką wykona kompilator.

Parser LR składa się z *wejścia*, *wyjścia*, *stosu*, *programu sterującego zachowaniem* i *tablicy analizatora*. Program analizatora wczytuje po kolei znaki z wejścia. Używa on stosu do zapamiętania ciągu o postaci $s_0X_1s_1X_2s_2\ldots X_ms_m$, z s_m na wierzchołku. Każde X_i jest symbolem gramatyki, a s_i jest symbolem nazywanym *stanem*. Każdy symbol stanu jest podsumowaniem informacji o stosie, a kombinacja symbolu stanu z wierzchołka stosu i aktualnego symbolu wejściowego jest używana do indeksowania tablicy analizatora, oraz do podejmowania decyzji o przesunięciu lub redukcji.

Tablica analizatora składa się z dwóch części, funkcji wyznaczającej akcje nazywanej *akcja* oraz funkcji wyznaczającej przejścia *przejście*. Program sterujące sprawdza stan na wierzchołku stosu Sm oraz aktualny symbol na wejściu a_i. Następnie odczytuje wartość *akcja*[s_m, a_i] w tablicy analizatora dla stanu s_m i wejścia a_i, która może być jedną z zaprezentowanych w Tabeli 8. Tabela ta zawiera bardziej precyzyjny opis akcji niż ten przedstawiony w Tabela 13.

Konfiguracją analizatora LR nazywamy parę, której pierwszym elementem jest zawartość stosu a drugim nieprzeczytane wejście.

$$(s_0X_1s_1X_2s_2\ \ldots\ X_ms_m,\ a_ia_{i+1},\ \ldots\ a_n\ \$)$$

Funkcja przejście bierze jako argumenty stan i symbol z gramatyki a zwraca stan. Funkcja *przejście* dla tablicy analizatora zbudowanej z gramatyki *g* jest funkcją przejścia deterministycznego automatu skończonego, który rozpoznaje prefiksy żywotne *G*.

Tabela 14 Możliwe akcje do wykonania przez parser wraz z ich formalnym opisem

AKCJA	OPIS DZIAŁANIA
akcja[s_m, a_i] = przesuń *s*	analizator wykonuje przesunięcie, przechodząc do konfiguracji $(s_0X_1s_1X_2s_2\ \ldots\ X_ms_ma_is,\ a_{i+1}\ \ldots\ a_n\ \$)$ analizator wstawia na stos aktualny symbol wejściowy a_i i następnie stan *s*, który jest pobierany z *akcja*[s_m, a_i], aktualnym symbolem wejściowym staje się a_{i+1}
akcja[s_m, a_i] = redukuj według A → B	analizator wykonuje redukcję przechodząc do konfiguracji $(s_0X_1s_1X_2s_2\ \ldots\ X_{m-r}s_{m-r}As,\ a_ia_{i+1}\ \ldots\ a_n\ \$)$ gdzie, s = *przejście*[s_{m-r}, A], a r jest długością β czyli prawej strony produkcji. analizator zdejmuje ze stosu 2r symboli, odkrywa stan s_{m-r}. wstawia na stos A – lewą stronę użytej produkcji i s – wartość *przejście*[s_{m-r}, A]. Aktualny symbol wyjściowy w wyniku redukcji nie jest zmieniany.
akcja[s_m, a_i] = akceptuj	analiza jest zakończona pomyślnie
akcja[s_m, a_i] = błąd	analiza jest zakończona błędem

Tablica analizatorów LR

Sytuacją LR(0) gramatyki *G* nazywamy produkcję *G* z kropką w jakimś miejscu jej prawej strony. Kropką zaznacza się, jaki moment produkcji już widzieliśmy w danym momencie procesu wyprowadzania.

Dla produkcji $A \rightarrow A_1 A_2 A_3$ można otrzymać następujące sytuacje

$$A \rightarrow \cdot A_1 A_2 A_3$$
$$A \rightarrow A_1 \cdot A_2 A_3$$
$$A \rightarrow A_1 A_2 \cdot A_3$$
$$A \rightarrow A_1 A_2 A_3 \cdot$$

W metodzie SLR konstruuje się z gramatyki deterministyczny automat skończony, który rozpoznaje prefiksy żywotne. Sytuacje łączy się w zbiory, z których powstają stany analizatora SLR.

Wzbogaconą gramatyką G' gramatyki *G* nazywamy, gramatykę *G* z nowym symbolem startowym *S*' i produkcją *S*' → *S*, gdzie *S* jest symbolem startowym gramatyki *G*. Celem wprowadzenia tej dodatkowej produkcji jest, ułatwienie parserowi wykrycia momentu końca analizy. Jeśli parser dokonuje redukcji *S* do *S*' oznacza to, że analiza będzie zakończona pomyślnie.
Formalnie, wzbogacona gramatyka G' jest uporządkowaną czwórką

$$G' = < N \cup \{S'\}, T, P \cup \{ S' \rightarrow S \}, S' >$$

Operacją *domknięcie*(*I*), gdzie *I* jest zbiorem sytuacji gramatyki *G* nazywamy zbiór sytuacji otrzymanych z *I* przy zastosowaniu poniższych reguł:

1. każda sytuacja z *I* jest dodawana do *domknięcie*(*I*)
2. jeśli $A \rightarrow \alpha \cdot B\beta$ jest w *domknięcie*(*I*), a $B \rightarrow \gamma$ jest produkcją, to – jeśli nie zostało to zrobione wcześniej – do *I* dodaje się sytuację $B \rightarrow \cdot\gamma$. Regułę tę stosuje się dopóki do *domknięcie*(*I*) nie można dodać żadnego nowego elementu.

Niech $A \rightarrow \alpha \cdot B\beta$ będzie w *domknięcie*(*I*). Oznacza to, że w pewnej chwili podczas prowadzenia analizy oczekuje się, iż na wejściu może być podciąg wyprowadzany z $B\beta$. Dla produkcji $B \rightarrow \gamma$ oczekuje się również, że w tej samej chwili na wejściu może być podciąg wyprowadzalny z γ. Z tego powodu do *domknięcie*(*I*) dołącza się również $B \rightarrow \cdot\gamma$.

(0) *wyrazenie'* → *wyrazenie*
(1) *wyrazenie* → *wyrazenie* opA *skladnik*
(2) *wyrazenie* → *skladnik*
(3) *skladnik* → *skladnik* opM *czynnik*
(4) *skladnik* → *czynnik*
(5) *czynnik* → liczba
(6) *czynnik* → (*wyrazenie*)

Rys. 4.33 Gramatyka dla wyrażeń matematycznych wraz z numeracją produkcji, **opA** oznacza operator addytywny (+-), **opM** oznacza operator multiplikatywny (*,/)

Jeśli *G* będzie gramatyką przedstawioną na rysunku 4.38, a *I* jednoelementowym zbiorem {[*wyrazenie* → · *wyrazenie*]}, to *domknięcie*(*I*) zawiera następujące sytuacje:

wyrazenie' → · *wyrazenie*
wyrazenie → · *wyrazenie* opA *skladnik*
wyrazenie → · *skladnik*
skladnik → · *skladnik* opM *czynnik*
skladnik → · *czynnik*
czynnik → · liczba
czynnik → · (*wyrazenie*)

Rys. 4.34 Sytuacje domknięcie {[*wyrazenie* → · *wyrazenie*]}

Funkcja *przejście*(*I*, *X*), gdzie *I* jest zbiorem sytuacji, a *X* jest symbolem z gramatyki, jest definiowana jako domknięcie zbioru wszystkich sytuacji $[A \rightarrow \alpha X \cdot \beta]$ takich, że $[A \rightarrow \alpha X \cdot \beta]$ jest w *I*.
Jeśli *I* jest zbiorem sytuacji, które są możliwe dla pewnego prefiksu żywotnego γ, to *przejście*(*I*, *X*) jest zbiorem sytuacji, które są możliwe dla prefiksu żywotnego γX.

wyrazenie → *wyrazenie* opA · *skladnik*
skladnik → · *skladnik* opM *czynnik*
skladnik → · *czynnik*
czynnik → · liczba
czynnik → · (*wyrazenie*)

Rys. 4.35 *przejście*({[*wyrazenie'* → *wyrazenie* ·], [*wyrazenie* → *wyrazenie* · opA *skladnik*]}, opA)

Niech *I* jest zbiorem dwóch sytuacji, {[*wyrazenie'* → *wyrazenie* ·], [*wyrazenie* → *wyrazenie* · opA *skladnik*]}, to *przejście*(*I*, opA) składa się z *przejście*(*I*, opA) oblicza się, wyszukując w *I* sytuacje, które mają opA bezpośrednio po prawej stronie kropki. *wyrazenie'* → *wyrazenie* · nie jest taką sytuacją, ale *wyrazenie* · opA *skladnik* jest. Kropkę przesuwa się za opA, otrzymując { *wyrazenie* → *wyrazenie* opA · *skladnik* } i obliczając domknięcie tego zbioru.

Mając zdefiniowane funkcje domknięcie i przejście, można podać algorytm budowania kanonicznej rodziny zbiorów sytuacji LR(0) dla wzbogaconej gramatyki *G*'.

```
void sytuacje( WzbogaconaGramatyka G' ){
      C = { domknięcie( { [ S' → ·S] } ) };
      do{
            for ( każdy zbiór sytuacji I z C i każdy symbol X z gramatyki
                  taki, że przejście(I, X) nie jest puste i nie jest w C
      )
            {
                  dodaj przejście(I, X) do C
            }
      } while ( do C nie można dodać żadnego nowego symbolu );
}
```

Rys. 4.36 Algorytm obliczania zbiorów sytuacji

Dla każdej gramatyki *G*, funkcja *przejście* dla kanonicznej rodziny zbiorów sytuacji definiuje deterministyczny automat skończony, który rozpoznaje żywotne prefiksy dla *G*.

Na Rys. 4.37 przedstawiono obliczone zbiory sytuacji dla gramatyki z Rys. 4.33.

Stan I_0 zawiera produkcje z Rys. 4.33, z tym, że każda z nich zaczyna się od kropki. Odpowiada to zbiorowi obliczonemu jako *domknięcie*[*S*' → ·*S*]. Dla zbioru I_0 oraz dla każdego symbolu gramatyki *X,* (*wyrazenie*, *skladnik*, *czynnik*, **opA**, **opM**, **(**, **)**), tworzy się

nowy zbiór obliczany jako *przejście*(I_0, X) i dodaje się go do C. Z *przejście*(I_0, *wyrazenie*) powstaje zbiór I_1, z *przejście*(I_0, *skladnik*) powstaje zbiór I_2, z *przejście*(I_0, czynnik) powstaje zbiór I_3, zbiór I_4 dopowiada *przejście*(I_0, ( ) natomiast *przejście*(I_0, **opA**) oraz *przeście*(I_0, **opM**) tworzą zbiory puste dlatego zgodnie z algorytmem nie dodaje się ich do C. Ponieważ dla zbioru I_0 obliczyliśmy przejścia dla wszystkich możliwych symboli gramatyki, procedurę tę powtarza się dla nowy powstałych zbiorów. Zbiór I_6 powstaje jako *przejście*(I_1, **opA**). Algorytm wykonuje się dopóki do C nie można dodać żadnego nowego symbolu.

Sytuacja $A \rightarrow \alpha_1 \cdot \beta_2$ nazywana jest *możliwą* dla prefiksu żywotnego $\alpha\beta_1$, jeśli istnieje wyprowadzenie $S' \Rightarrow_{rm}^{*} \alpha Aw \Rightarrow_{rm}^{*} \alpha\beta_1\beta_2 w$. Fakt, że sytuacja $A \rightarrow \alpha_1 \cdot \beta_2$ jest możliwa dla $\alpha\beta_1$, mówi o tym, czy powinno wykonać się przesunięcie czy redukcję, gdy na stosie analizatora znajduje się $\alpha\beta_1$. Jeśli $\beta_2 = \varepsilon$, $A \rightarrow \beta_1$ jest uchwytem i można go zredukować przy użyciu tej produkcji. W przeciwnym wypadku, jeśli $\beta_2 \neq \varepsilon$, jeszcze nią ma uchwytu na stosie i należy wykonać przesunięcie.

I_0
wyrazenie → · *wyrazenie*
wyrazenie → · *wyrazenie* **opA** *skladnik*
wyrazenie → · *skladnik*
skladnik → · *skladnik* **opM** *czynnik*
skladnik → · *czynnik*
czynnik → · (*wyrazenie*)
czynnik → · **liczba**

I_1
wyrazenie → *wyrażenie* ·
wyrazenie → *wyrazenie* · **opA** *skladnik*

I_2
wyrazenie → *skladnik* ·
skladnik → *skladnik* · **opM** *czynnik*

I_3
skladnik → *czynnik* ·

I_4
czynnik → (· *wyrazenie*)
wyrazenie → · *wyrazenie* **opA** *skladnik*
wyrazenie → · *skladnik*
skladnik → · *skladnik* **opM** *czynnik*
czynnik → · (*wyrazenie*)
czynnik → · **liczba**

I_5
czynnik → **liczba** ·

I_6
wyrazenie → *wyrazenie* **opA** · *skladnik*
skladnik → · *skladnik* **opM** *czynnik*
skladnik → · *czynnik*
czynnik → · (*wyrazenie*)
czynnik → · **liczba**

I_7
skladnik → *skladnik* **opM** · *czynnik*
czynnik → · (*wyrazenie*)
czynnik → · **liczba**

I_8
czynnik → (*wyrazenie* ·)
wyrazenie → *wyrazenie* · **opA** *skladnik*

I_9
wyrazenie → *wyrazenie* **opA** *skladnik* ·
skladnik → *skladnik* · **opM** *czynnik*

I_{10}
skladnik → *skladnik* **opM** *czynnik* ·

I_{11}
czynnik → (*wyrazenie*) ·

Rys. 4.37 Zbiory sytuacji dla gramatyki Rys. 4.33 [2]

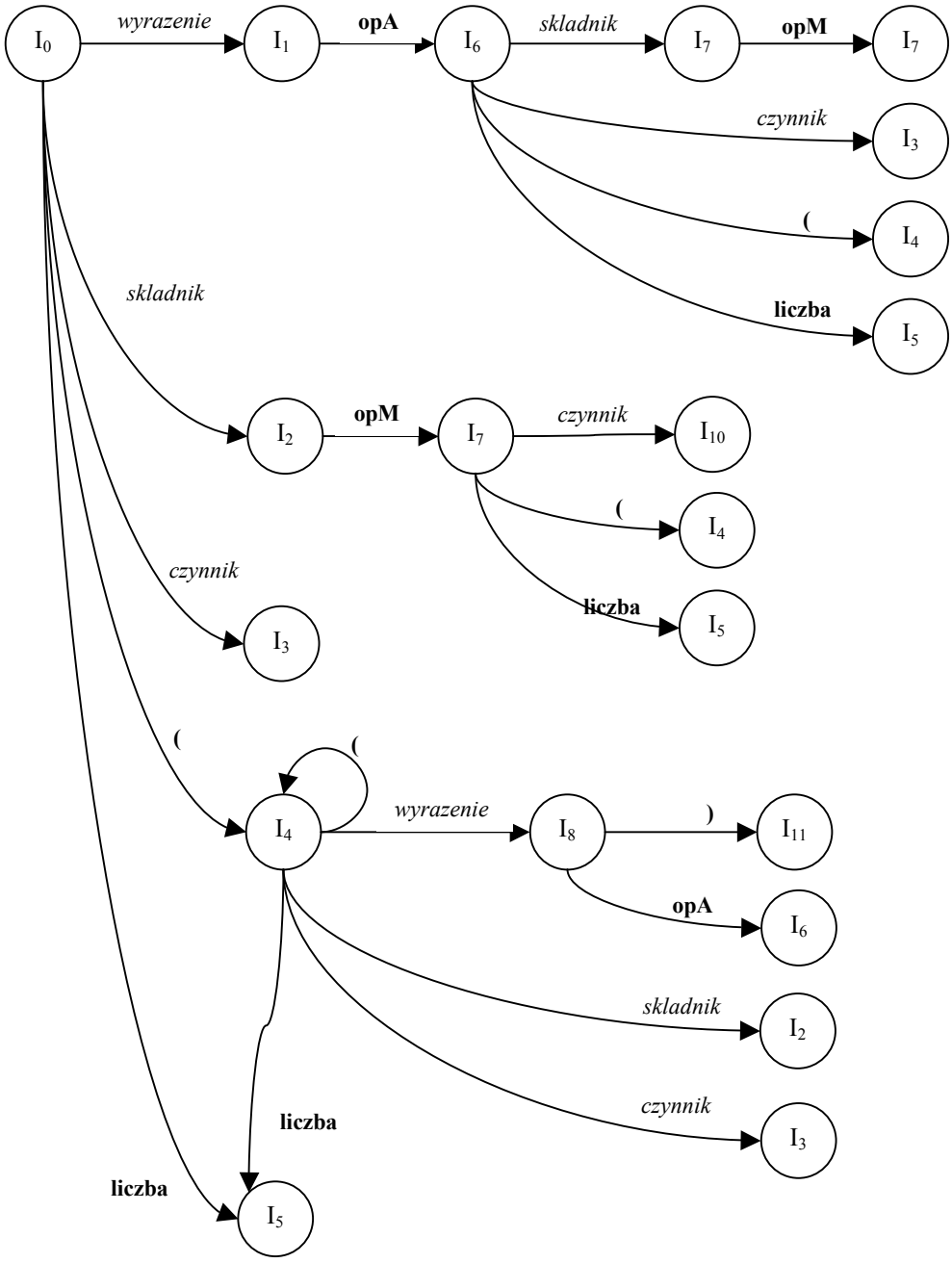
Na zaprezentowano algorytm budowy tablicy analizatora SLR. Nazwa SLR pochodzi od Simple LR, czyli prosty LR.

1. Zbudować $C = \{ I_0, I_1, \ldots, I_n \}$, rodzinę zbiorów sytuacji LR(0) dla G'
2. Stan i buduje się z I_i, Akcje analizatora dla stanu i wyznacza się następująco
    - jeśli $[A \rightarrow \alpha \cdot a\beta]$ jest w I_i oraz *przejście*$(I_i, a) = I_j$, to elementowi *akcja*$[i, a]$ nadaje się wartość „*przesuń j*", gdzie a musi być terminalem
    - jeśli $[A \rightarrow \alpha \cdot]$ jest w I_i, to elementowi *akcja*$[i, a]$ nadaje się wartość „*redukuj według* $A \rightarrow \alpha$ " dla wszystkich a z FOLLOW(A), A nie może być równe S'
    - jeśli $[S' \rightarrow S]$ jest w I_i, to elementowi *akcja*$[i, \$]$ nadajemy wartość „*akceptuj*"
3. Wartości przejście dla stanu i są tworzone dla wszystkich nieterminali A zgodnie z regułą, jeśli *przejście* $(I_i, A) = I_j$, to *przejście*$[i, A] = j$
4. Wszystkie pozycje tablicy, którym w krokach 2 i 3 nie nadano wartości oznacza się jako błędne
5. Stan startowy analizatora to stan zbudowany ze zbioru sytuacji, do którego należy sytuacja $[S' \rightarrow S]$

Rys. 4.38 Algorytm budowy tablicy SLR [2]

Niech dany będzie zbiór sytuacji I_0 przedstawiony na Rys. 4.37.
Sytuacja *czynnik* → · (*wyrazenie*) powoduje powstanie wpisu *akcja*[0, (] = *przesuń 4*, natomiast sytuacja *czynnik* → · liczba wpisu *akcja*[0, id] = *przesuń* 5. Pozostałe sytuacje ze zbioru I_0 nie generują żadnych wpisów do tablicy. Zbiór *I*1 na rysunkuRys. 4.39 dodaje nowe wpisy do tablicy. Sytuacja *wyrazenie* → *wyrażenie* · dodaje *akcja*[1, $] = *akceptuj*, druga *wyrazenie* → *wyrazenie* · opA *skladnik akcja*[1, opA] = *przesuń* *6*.
Zbiór I_2 zawiera dwie stuacje. Ponieważ FOLLOW(*wyrazenie*) = {$, opA,)}, pierwsza sytuacja daje akcja[2, $] = akcja[2, opA] = akcja[2,)] = *redukuj* według *wyrazenie* → *skladnik,* druga sytuacja daje *akcja*[2, opM] = *przesun 7*.

Algorytm budowania tej tablicy nie jest skomplikowany, jednak metody SLR nie można użyć do parsowania wszystkich instrukcji programistycznych. O wiele większe możliwości ma bardziej złożona oraz pamięciożerna kanoniczna tablica LR. Algorytm budowy kanonicznej tablicy przedstawiono w pozycji [2].



Rys. 4.39 Diagram przejść dla gramatyki z Rys. 4.33 [2]

Tabela 15 Tablica analizatora wstępującego dla gramatyki z Rys. 4.33 [2]

STAN	*akcja*						*przesunięcie*		
	liczba	**opA**	**opM**	**(**	**)**	**$**	*wyrazenie*	*skladnik*	*czynnik*
0	s5			s4			1	2	3
1		s6				akc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		r6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Algorytm analizy LR

Wejściem dla algorytmu analizy LR są, ciąg wejściowy w oraz tablica analizatora LR z funkcjami akcja i przejście dla gramatyki *G*.
Wyjściem jest prawostronne wyprowadzenie dla *w* jeśli *w* jest w *L*(*G*), lub informacja o błędzie w przeciwnym razie.
Początkowo na stosie analizatora jest s_0, czyli stan początkowy, a na wejściu jest *w*$. Analizator wykonuje program z rysunku 4.44 aż do napotkania akcji *akceptuj* lub *błąd*.
Przykład zawartości stosu, wejścia oraz akcji wykonywanych przez parser podczas parsowania napisu *t* = **liczba** * **liczba** + **liczba** zaprezentowano na rysunku 4.46.
Na początku analizy składniowej stos parsera jest pusty 0, a bieżącym symbolem jest **liczba**. Zgodnie z akcją w tablicy analizatora, na pozycji *akcja*[0, **liczba**] znajduje się *s*5, czyli przesuń (ang. *shift*) bieżący symbol na stos a na nim połóż stan 5. Po tym kroku na stosie parsera znajduje się 0 liczba 5 a bieżącym symbolem staje się operator multiplikatywny * czyli **opM**. W tablicy analizatora, na pozycji akja[5, **opM**] znajduje się wpis r6, czyli redukuj (ang. *reduce*) zgodnie z regułą gramatyczną 6. Gramatyka wraz z ponumerowanymi regułami znajduje się na rysunku 4.38. Regułą tą jest *czynnik* → *składnik*, przy czym prawa strona produkcji zawiera jeden symbol. W wyniku tego ze stosu zdejmowane są 2*|skladnik| symboli, czyli dwa symbole (5, *czynnik*), wykonywana jest redukcja **liczby** do *czynnika* i *czynnik* wkładany jest z powrotem na stos. Następnie parser przechodzi do stanu 3, ponieważ *przesunięcie*(0, *czynnik*) wynosi 3. Stos parsera w tej chwili zawiera 0 czynnik 3. Dalsze kroki parsowania wykonywane są w analogiczny sposób.

```
zainicjuj ip tak, aby wskazywało pierwszy symbol w$
while ( true ) {
      niech s będzie symbolem z wierzchołku stosu
            i a symbolem wskazywanym przez ip;

      if akcja[s, a] = przesuń s'{
            wstaw a, następnie s' na wierzchołek stosu;
            przesuń ip do następnego symbolu wejściowego;
      }
      else if akcja[s, a] = redukuj według A → β{
            zdejmij ze stosu 2*|β| symboli;
            niech  s' będzie stanem, który znalazł się na
            wierzchołku stosu;
            wstaw  A i przejście[s', A] na wierzchołek stosu;
            wypisz produkcję A → β
      }
      else if akcja[s, a] = akceptuj{
            return AnalizaZakonczonaPowodzeniem;
      }
      else{
            return Błąd;
      }
}
```

Rys. 4.40 Program wstępującego parsera [2]

	STOS	WEJŚCIE	AKCJA
(1)	0	**liczba** * **liczba** + **liczba** $	przesunięcie
(2)	0 **liczba** 5	* **liczba** + **liczba** $	*czynnik* → **liczba**
(3)	0 *czynnik* 3	* **liczba** + **liczba** $	*skladnik* → *czynnik*
(4)	0 *skladnik* 2	* **liczba** + **liczba** $	przesunięcie
(5)	0 *skladnik* 2 * 7	**liczba** + **liczba** $	przesunięcie
(6)	0 *skladnik* 2 * 7 **liczba** 5	+ **liczba** $	*czynnik* → **liczba**
(7)	0 *skladnik* 2 * 7 *czynnik* 10	+ **liczba** $	*skladnik* → *skladnik* **opM** *czynnik*
(8)	0 *skladnik* 2	+ **liczba** $	*wyrazenie* → *skladnik*
(9)	0 *wyrazenie* 1	+ **liczba** $	przesunięcie
(10)	0 *wyrazenie* 1 + 6	**liczba** $	przesunięcie
(11)	0 *wyrazenie* 1 + 6 **liczba** 5	$	*czynnik* → **liczba**
(12)	0 *wyrazenie* 1 + 6 *czynnik* 3	$	*skladnik* → *czynnik*
(13)	0 *wyrazenie* 1 + 6 *skladnik* 9	$	*wyrazenie* → *wyrazenie* **opA** *skladnik*
(14)	0 *wyrazenie* 1 +	$	akceptowanie

Rys. 4.41 Zawartość stosu, symbole na wejściu oraz podejmowane akcje podczas parsowania napisu **liczba * liczba + liczba**

5 Analiza semantyczna

Analiza semantyczna jest ostatnią fazą przedniej części kompilacji. Jej zadaniem jest wykrycie błędów niemożliwych do znalezienia podczas analizy leksykalnej i składniowej. Typowe błędy analizy semantycznej pokazano w Tabela 16.

Tabela 16 Zadania fazy semantycznej wraz z typowymi błędami

ZADANIE ANALIZY SEMANTYCZNEJ	OPIS TYPOWYCH BŁEDÓW
Kontrola typów	Użycie funkcji/operatora dla niepoprawnych typów, dodawanie napisu i liczby Zgodność liczby i typów parametrów w wywołaniach funkcji
Kontrola przepływu sterowania	Użycie instrukcji *continue* poza ciałem pętli
Kontrola zasięgu widoczności	Każda zmienna może być zdefiniowana w jednym zasięgu tylko raz, Można odwołać się do zadeklarowanych zmiennych

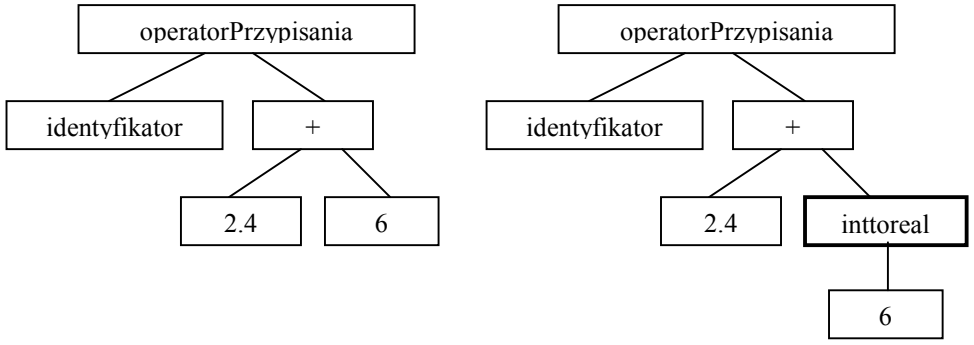
Wejściem dla analizy semantycznej jest drzewo składniowe dostarczone przez parser. Nie jest wygodne mieszanie kodu, który buduje to drzewo jak również sprawdza jego poprawność logiczną. Drzewo składniowe, chociaż może być błędne semantycznie jest na pewno zgodne z gramatyką. W wielu przypadkach parser ma za mało informacji, żeby wykryć błąd.

Przykład 5.1
Deklaracje zmiennej od jej użycia może dzielić wiele linii kodu. Mogą to być często również dwa zupełnie osobne pliku.

```
prędkośćObrotowąSilnika = 0 ;
...
unsigned const prędkośćObrotowąSilnika = 2000;
```

Aby wykryć, że składniowo poprawna instrukcja jest przypisaniem na zmienną stałą należy odnaleźć deklarację tej zmiennej wraz z uwzględnieniem całego mechanizmu wyszukiwania nazw lokalnych.□

Wyjściem analizy semantycznej jest drzewo składniowe wzbogacone o węzły z instrukcjami semantycznymi oraz dodatkowe informacje pomocne w generacji kodu ostatecznego takie jak tablica symboli. Przykład drzewa z instrukcjami semantycznymi pokazano na Rys. 5.1.



Rys. 5.1 Drzewo składniowe z instrukcjami semantycznymi

5.1 Tablica symboli

Ważną funkcją kompilatora jest zapamiętywanie identyfikatorów używanych w programie źródłowym i zbieranie informacji o tych identyfikatorach. Informacje te mogą określać typ zmiennej, czy zmienna jest stałą, jej zasięg widoczności i wiele innych rzeczy. Oprócz zmiennych, za słowem identyfikator może kryć się również nazwa typu zdefiniowanego przez użytkownika jak również funkcji. Przykładowe atrybuty dla zmiennej, funkcji oraz klasy zaprezentowano w Tabela 17.

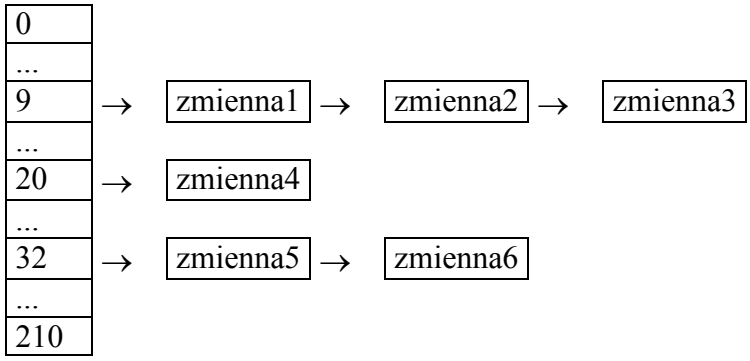
Tabela 17 Atrybuty dla zmiennej, funkcji, klasy

ZMIENNA	NAZWA FUNKCJI	NAZWA KLASY
nazwa	nazwa	nazwa
typ	ilość parametrów	klasy bazowe
czy jest stałą (*const*, *mutable*)	czy jest stałą w przypadku metod (*const*)	pola składowe
czy dostęp jest optymalizowany (*volatile*)	typu parametrów	funkcje składowe
położenie w pamięci	typ wartości zwracanej	czy jest klasą czy strukturą (*class*, *struct*)
zasięg widoczności	czy jest wirtualna (*virtual*)	czy jest klasą abstrakcyjną
prawa dostępu dla atrybutów w klasie (*public*, *private*, *protected*)	czy jest czystowirtualna (*=0*)	

Implementacja tablicy symboli w oparciu o algorytmy mieszania

Tablica symboli może być zaimplementowana przy pomocy takich struktura danych jak tablica, lista, lista dwukierunkowa itp. Częste operacje wyszukiwania symboli w takiej tablicy, wymuszają stosowanie struktur danych optymalizowanych ze względu na wyszukiwanie jak również optymalnych algorytmów. Bardzo dobre efekty wydajnościowe wyszukiwania w tablicy symboli można uzyskać stosując technikę przeszukiwania zwaną mieszaniem (ang. *hashing*). Zastosowanie mieszania daje możliwość wykonania *e* zapytań na *n* nazwach w czasie proporcjonalnym do *n(n+e)/m*, dla dowolnie wybranej stałej *m*. Ponieważ *m* może być dowolnie duże, aż do *n*, ta technika jest ogólnie bardziej wydajna niż lista i jest najczęściej wybierana dla tablicy symboli. Złożoność pamięciowa algorytmu jest wprost proporcjonalna do stałej *m*, należy więc wybrać pomiędzy złożonością czasową a pamięciową.

Tablicę mieszającą o rozmiarze 211 przedstawiono na Rys. 5.2.



Rys. 5.2 Tablica mieszająca o rozmiarze 211

Tablica mieszająca składa się z dwóch części

1. tablica mieszającej, będącej stałej długości tablica wskaźników do poszczególnych wpisów
2. wpisów zorganizowanych w *m* oddzielnych listach jednokierunkowych.

Każdy wpis w tablicy symboli znajduje się dokładnie w jednej z takich list. O tym w której liście znajduje się wpis decyduje funkcja mieszająca (ang. *hashing function*). Funkcja mieszająca *h* odwzorowuje nazwę symbolu *s* na dowolną liczbę całkowitą *c*. Aby odwzorować liczbę *c* do przedziału od 0 do $m - 1$ często dzieli się ją przez rozmiar tablicy *m,* a reszta z dzielenia daje pożądany wynik. Z tego powodu, zaleca się, aby tablica mieszająca miała rozmiar będący liczbą pierwszą. Jeśli *s* znajduje się w tablicy symboli to jest na liście numer *h(s)*. Przeszukiwanie listy *h(s)* odbywa się liniowo. Jeśli w tablicy o rozmiarze *m* znajduje się *n* wpisów, to pojedyncza lista ma średnio *n/m* wpisów. Wybierając odpowiednio duże *m*, przeszukiwanie list może odbywać się bardzo szybko.

Znalezienie dobrej funkcji mieszającej nie jest zadaniem prostym. Według P. J. Weinbergera [6] bardzo dobre rezultaty daje funkcja mieszająca *hashpjw*, której implementacja zaprezentowana jest na Rys. 5.3.

```
int hashpjw( std::string const & _symbol, unsigned const _range ){
      unsigned hash = 0;
      unsigned value = 0;

      for ( std::string::const_iterator
                  current ( _symbol.begin() )
                , end( _symbol.end() )
            ; current != end
            ; ++current
      ){
            hash = (hash << 4 ) + (*current);
            if (  value = hash & 0xf0000000 ){
                  hash = hash ^ (value >> 24 );
                  hash = hash ^ value;
            }
      }
      return hash % _range;
}
```

Rys. 5.3 Funkcja hashpjw [6]

Tablica symboli może zawierać słowa kluczowe (ang. *keyword*). Należy ją wtedy zainicjować przed pierwszym użyciem wpisami odpowiadającymi słowom kluczowym. Technika ta może być stosowana, jeżeli w jakichś powodów słowa kluczowe nie mogą być elementem gramatyki jako symbole terminalne.

5.2 Zasięg widoczności zmiennych

Zalecaną techniką jest deklarowanie zmiennych o jak najmniejszym zasięgu. Prawdopodobieństwo pomyłki jest wtedy mniejsze, można również wykorzystać nazwę tej zmiennej wielokrotnie w różnych zasięgach.

Przykład 5.2
Język C++ dopuszcza deklarowanie zmiennych w jak najbardziej lokalnych zasięgach. Typowym przykładem jest pętla `for`, której pierwsza instrukcja odpowiada opcjonalnej deklaracji zmiennej indeksującej. Zmienną zadeklarowaną w ten sposób można używać tylko w ciele pętli.

```
for ( typZmiennej nazwaZmiennej ; warunek ; instrukcja ) {
      // treść pętli
}
```

Innym przykładem, jest możliwość deklarowania zmiennych w instrukcji warunkowej `if`.

```
if ( typZmiennej nazwaZmiennej , warunek ) {
      // instrukcje wykonywane, gdy warunek jest prawdziwy
}
else {
      // instrukcje wykonywane, gdy warunek jest fałszywy
}
```

Zmienna zadeklarowana w ten sposób jest widoczna tylko w obydwóch gałęziach instrukcji warunkowej.□

Konstrukcją gramatyczną, która pozwala grupować zmienne w logiczne grupy są bloki zwane również zasięgami (ang. *scope*). Ich charakterystyczną cechą jest możliwość zagnieżdżania. Jeden zasięg może zawierać się w innym, lecz nie jest możliwe nakładanie się bloków na siebie.

Język C++ pozwala na definiowanie zmiennej o takiej samej nazwie (*przesłanianie*), pod warunkiem, że definicje te miały miejsce w różnych blokach.

Widzialność deklaracji w języku o strukturze blokowej jest określona przez regułę *najbliższego zagnieżdżenia*.

1) Deklaracja z bloku *B* jest widzialna w bloku *B*
2) Jeśli nazwa *x* nie została zadeklarowana w bloku *B*, to wystąpienie x w bloku *B* znajduje się w zakresie widzialności deklaracji x w bloku *B'* zawierającym blok *B*, takim, że
    a) *B'* zawiera deklarację *x*
    b) *B'* względem zagnieżdżenia jest najbliżej bloku *B*, bliżej niż każdy inny blok z deklaracją *x*.

Niech dany będzie program napisany w języku C++.

```
int main( int argc )
{ // blok B0
      int a = 0;
      int b = 0;

      { // blok B1
            int b = 1;

            { // blok B2
                  int a = 2;
                  std::cout << a << b << std::endl;
            } // blok B2

            { // blok B3
                  int b = 3;
                  std::cout << a << b << std::endl;
            } // blok B3
      } // blok B1

      // int argc
      // w tym miejscu deklaracja argc jest błędna, ponieważ
      // argc zostało już zadeklarowane jako parametr funkcji

      std::cout << a << b << std::endl;
} // koniec bloku B0
```

Zakres widoczności poszczególnych zmiennych zaprezentowano w Tabela 18.

Tabela 18 Zakres widoczności zmiennych w programie

DEKLARACJA	WIDZIALNOŚĆ
`int argc`	B_0
`int a = 0;`	$B_0 - B_2$
`int b = 0;`	$B_0 - B_1$
`int b = 1;`	$B_1 - B_3$
`int a = 2;`	B_2
`int b = 3;`	B_3

W języku C++ obowiązują również dodatkowe mechanizmy do definiowania widzialności. Najbardziej popularnym są przestrzenie nazw (ang. *namespace*). Przestrzeń nazw jest nazwanym blokiem, który może rozciągać się na przestrzeni wielu plików. Przestrzeń nazw pozwala grupować nazwy w logiczne jednostki, zabezpieczając w dużych projektach pisanych przez wielu programistów przed przypadkowym użyciem zdefiniowanych już symboli. Przeszukując przestrzenie nazw w poszukiwaniu użytych w kodzie symboli, wykorzystywana jest reguła wyszukiwania Koeninga (ang. *argument dependent lookup, ADL*) . [12]

Początkującym programistom zaskakujące efekty wyszukiwania nazw może również dać specjalizacja szablonu i używanie w niej nazw zależnych oraz niezależnych od typu szablonu. [19]

W języku Java, w celu uproszczenia mechanizmów językowych a także jako zabezpieczenie przez przypadkowymi błędami programisty, zabronione jest przesłanianie nazw.

5.3 Implementacja metod w języku C++

Każda metoda klasy reprezentowana jest w pamięci programu przez pojedynczą funkcję. W związku z możliwością definiowania metod o takiej samej nazwie, lecz różnej sygnaturze (ang. *overloading*) nazwy metod są kodowane. Schemat kodowania jest zależny od implementacji kompilatora. Wywołanie metody *m* na rzecz obiektu *o* jest tłumaczone przez kompilator na wywołanie funkcji *m'* , a położenie obiektu *o* jest przekazywane jako argument.

Przykład 5.3
Niech dana będzie klasa C zdefiniowana poniżej.

```
struct C {
      int funkcja( int _liczba );
};
```

Metoda klasy `C`, `funkcja`, zostanie zamieniona przez kompilator na zwykłą funkcję, której pierwszym parametrem jest wskaźnik na obiekt klasy `C`.

```
struct C {
};

int __zakodowana_nazwa_funkcja( C* this, int _liczba );
```

Wywołanie funkcji funkcja na rzecz obiektu klasy `C`

```
C c;
c.funkcja( 1 );
```

zostanie rozwinięte do następującego kodu.

```
C c;
__zakodowana_nazwa_funkcja ( &c, 1 );
```

Wskaźnik *this* jest zazwyczaj pierwszym argumentem wywołania metody. Dzięki niemu uzyskuje się dostęp do składowych klasy `C`. Wszelkie niekwalifikowane odwołania do składowych wewnątrz metody są niejawnie kwalifikowane wskaźnikiem *this*. Przedstawiony schemat nie znajduje jednak zastosowania do funkcji wirtualnych.

Implementacja wywołania metod wirtualnych

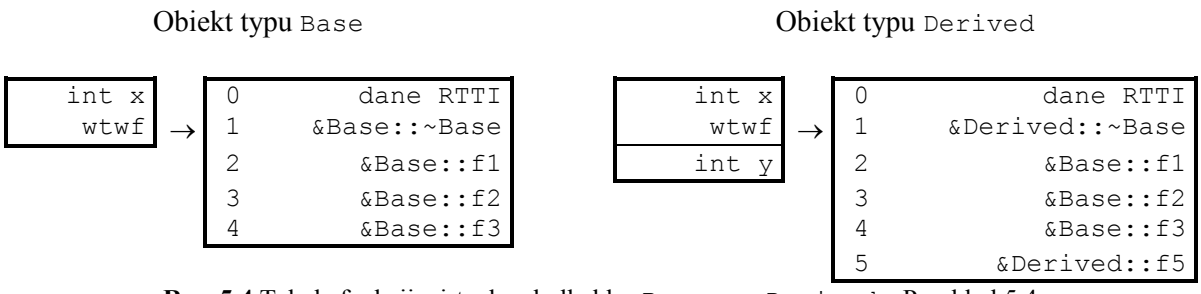
Język C++ udostępnia mechanizm klas wraz w implementacją funkcji wirtualnych (ang. *virtual functions*). Funkcja wirtualna zachowuje się w sposób polimorficzny. Oznacza to, że odwołując się do funkcji wirtualnej przez referencje lub wskaźnik, wywoła się funkcja właściwa dla rzeczywistego obiektu, na który referencja lub wskaźnik jest ustawiona.
Nie jest możliwe określenie wiązania funkcji wirtualnej podczas kompilacji. Wskazanie referencji lub wskaźnika, może zmieniać się dynamicznie podczas działania programu. W konsekwencji, odnalezienie typu obiektu, na który wskazuje referencja lub wskaźnik oraz odpowiadającej jemu definicji funkcji wirtualnej musi odbyć się podczas działania programu. Jest również możliwe wywołanie funkcji wirtualnej dla obiektu przechowywanego przez wartość, lecz wtedy nie obowiązują zasady polimorficznego wywołania funkcji.

W większości implementacji kompilatorów, do uzyskania polimorficznego zachowania używa się tabel funkcji wirtualnych oraz wskaźników do tych tabel.
Tabela funkcji wirtualnych jest tablicą wskaźników do funkcji. Dla każdej klasy w programie istnieje tabela jej funkcji wirtualnych. Każdy wpis w tabeli jest wskaźnikiem do implementacji funkcji wirtualnych tej klasy.

Przykład 5.4
Niech zdefiniowana będzie następująca hierarchia klas.

```
class Base{
public:
      Base();
      virtual ~Base();

      virtual void f1();
      virtual int f2( int );
      virtual double f3( double );

      char f4( char );

private:
      int x;
};

class Derived{
public:
      Derived();
      virtual ~Derived() ;

      virtual float f5( float ) ;

private:
      int y;
};
```

Tabela funkcji wirtualnych dla klas `Base` oraz `Derived` przedstawiona jest na Rys. 5.4. Tabela funkcji wirtualnych nie zawiera wpisów dla funkcji niewirtualnych, w tym statycznych oraz konstruktorów. Klasa dziedzicząca, zawiera wpisy dla nowych funkcji wirtualnych, które definiuje, oraz dla wszystkich odziedziczonych po klasach bazowych.□



Rys. 5.4 Tabela funkcji wirtualnych dla klas `Base` oraz `Derived` z Przykład 5.4

Istnieje kilka rozwiązań, co do przechowywania tablicy funkcji wirtualnych. Najprostsze polega na tym, że każda jednostka translacji zawiera kopię tej tablicy, natomiast podczas linkowania program konsolidujący usuwa zdublowane wpisy. Innym rozwiązaniem jest użycie heurystyki. Tabelę funkcji wirtualnych dla danej klasy tworzy się w pliku wynikowym zawierającym definicję pierwszej nierozwijalnej (nieoznaczonej słowem kluczowym *inline* lub nie zdefiniowanej w ciele klasy) funkcji składowej.

Każda klasa zawierająca funkcje wirtualne posiada jedną kopię tabeli funkcji wirtualnych, natomiast każdy obiekt klasy z funkcjami wirtualnymi, posiada jedno dodatkowe niejawne pole, które wskazuje na tablicę funkcji wirtualnych. W początkowych implementacjach wskaźnik tablicy umieszczany był jako ostatnie pole w obiekcie. Zachowano dzięki temu zgodność z językiem C, w którym adres pierwszego pola struktury jest adresem całej struktury. W przypadku wielodziedziczenia, takie rozwiązanie nie jest jednak efektywne. O wiele bardziej wydajne jest umieszczenie wskaźników do tablic wirtualnych klas bazowych na początku obiektu, tracąc na zgodności z językiem C. Obecnie standard języka nie narzuca sposobu ulokowania wskaźników do tablic funkcji wirtualnych pozwalając kompilatorom na optymalizację.

W celu wywołania funkcji wirtualnej kompilator tworzy następujący kod.
1. Dla danego obiektu znajduje tabelę funkcji wirtualnych wskazywaną przez wskaźnik tabeli funkcji wirtualnych.
2. W tabeli funkcji wirtualnych znajduje wskaźnik odpowiadający funkcji, która ma być wywołana.
3. Wywołuje funkcję wskazywaną przez odnaleziony wskaźnik.

Niech tabela funkcji wirtualnych będzie oznaczona jako *tfw*, a wskaźnik do niej *wtfw*. Wywołanie funkcji wirtualnej jest wtedy mniej więcej równoważne sekwencji przedstawionej na Rys. 5.5.

```
// wywołaj funkcję wskazywaną przez pozycję indeksFunkcji
// w tabeli tfw, na którą wskazuje referencjaLubWskaźnik->wtfw

      (*referencjaLubWskaźnik->wtfw[ indeksFunkcji ] )();
```

Rys. 5.5 Pseudokod wywołania funkcji wirtualnej

```
                                              ;skopiuj tfw do rejestru eax
      movl (object_address + wtwf_offset), %eax
                                    ; skopiuj adres funkcji do rejestru ebx
      movl (%eax + called_function_offset), %ebx
                                                          ; wywołaj funkcje
      call *ebx
```

Rys. 5.6 Kod języka asemblera dla wywołania funkcji wirtualnej

Na Rys. 5.6 widoczne jest, że mechanizm wywołania funkcji wirtualnych jest mechanizmem wydajnym. Powoduje on niewielki narzut kilku instrukcji maszynowych, zwalniając tym samym programistę od ręcznego śledzenia i kodowania rzeczywistych typów wskazywanych przez referencje i wskaźniki obiektów.

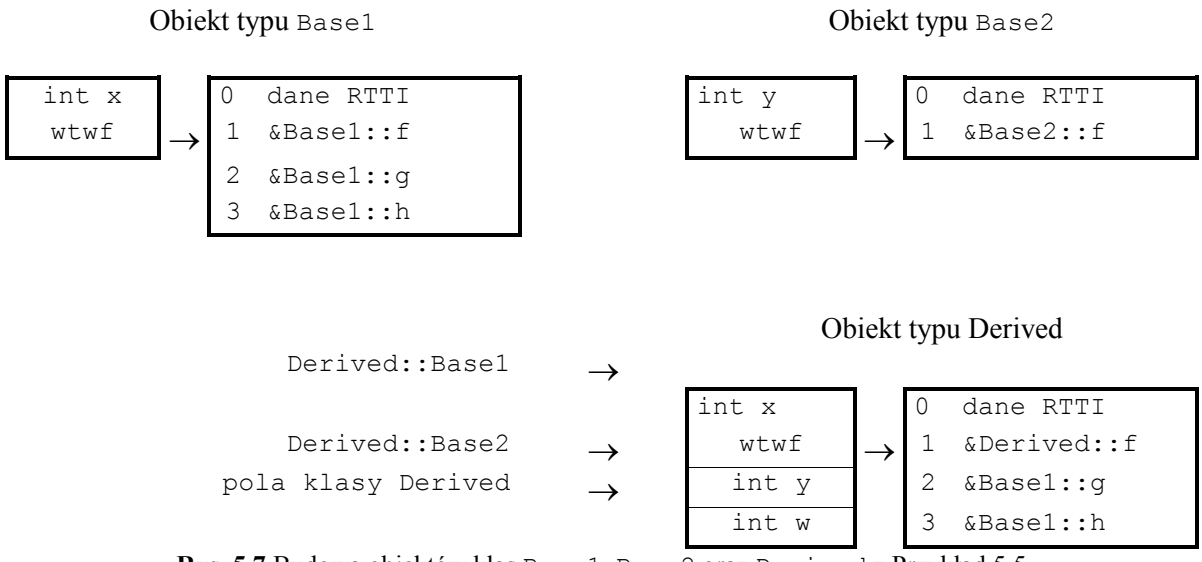
Implementacja wywołania metod w wielobazowej hierarchii dziedziczenia

W przypadku wielodziedziczenia mechanizm tablicy funkcji wirtualnych musi być bardziej ogólny, gdyż odnosi się do wielu klas bazowych zamiast do jednej. Budowę obiektu klasy w dziedziczeniu wielobazowym pokazano w Przykład 5.5 W przypadku dziedziczenia wielobazowego należy rozpatrzyć dwa przypadki, kiedy klasa `Derived` jest wykorzystywana jako obiekt `Base1` oraz kiedy klasa `Derived` jest wykorzystywana jako obiekt klasy `Base2`.

Przykład 5.5
Niech dana będzie hierarchia klas.

```
class Base1{
public:
      virtual void f() ;
      virtual void g() ;
      virtual void h() ;

private:
      int x;
};

class Base2 {
public:
      void ff() ;
      virtual void f();

private:
      int y;
};

class Derived2 : public Base1, public Base2 {
public:
      virtual void f();
      void hh();

private:
      int w;
};
```

Budowa obiektów klas `Base1`, `Base2` oraz `Derived` przedstawiona jest na Rys. 5.7.



Rys. 5.7 Budowa obiektów klas `Base1`, `Base2` oraz `Derived` z Przykład 5.5

W wywołaniu metody niewirtualnej `ff` przez wskaźnik ustawiony na obiekt klasy `Derived` pojawia się następujący problem.

```
Derived pD = new Derived;
pD->ff();
```

Wołana metoda niewirtualna `Base2::ff()` oczekuje jako jednego z argumentów wskaźnika *this* obiektu klasy `Base2`, a nie obiektu klasy `Derived`. Adres obiektu klasy `Derived` jest identyczny z adresem obiektu klasy `Base1`, ale obiekt klasy `Base2` jest przesunięty względem początku obiektu klasy `Derived` o pewien offset. Metoda `Base2::ff()` musi otrzymać wskaźnik `this`, który zawiera adres podobiektu klasy `Base2` klasy `Derived`. Przesunięcie podobiektu `Base2` względem początku obiekty klasy `Derived` jest znane w czasie kompilacji, dlatego w prosty sposób można zaimplementować takie wywołanie podczas kompilacji nie obciążając programu kosztami czasu wykonania. Kompilator musi jedynie dodać do adresu początku obiekty klasy `Derived` przesunięcie obiektu `Base2` i skonwertować otrzymany adres do typu `Base2*`. Dzięki tym operacjom niewirtualna metoda `D1::ff()` otrzymuje jako argument poprawną wartość wskaźnika `this`.

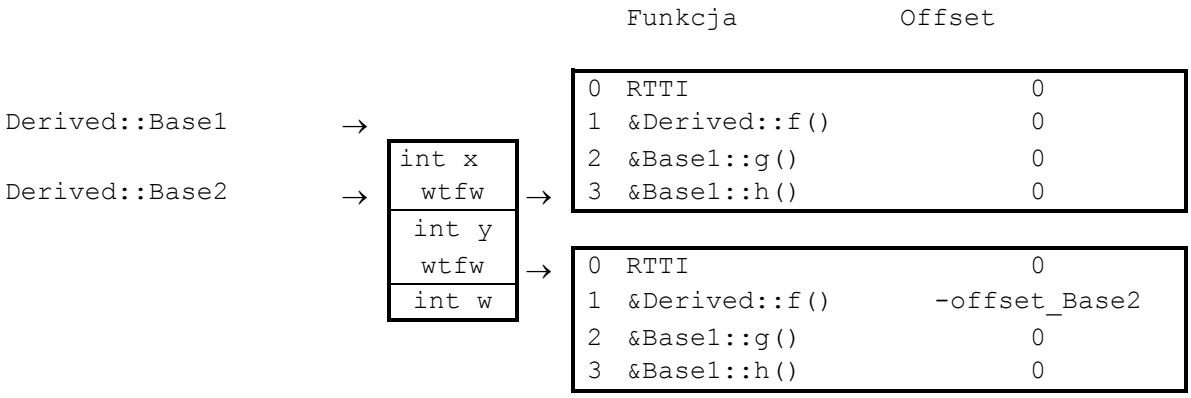
Implementacja wywołania metod w wielobazowej hierarchii dziedziczenia

Obliczanie wzajemnego położenia podobiektów względem siebie komplikuje się w hierarchii klas z wielodziedziczeniem.

```
Derived pDerived = new Derived;

// pBase1 musi wskazywać podobiekt Base1 w obiekcie Derived
Base1 pBase1 = new Base1;

// pBase2 musi wskazywać podobiekt Base2 w obiekcie Derived
Base2 pBase2 = new Base2;

// wywołuje Derived::f()
pDerived->f();

// wywołuje Derived::f(), chociaż pBase1 wskazuje na Base1
pBase1->f();            // wywołuje Base1::f()

// wywołuje Derived::f(), chociaż pBase2 wskazuje na Base2
pBase2->f();            // wywołuje Base2::f()
```

W zaprezentowanym kodzie wszystkie trzy wskaźniki `pDerived`, `pBase1` oraz `pBase2` odwołują się do różnych fragmentów obiektu klasy `Derived`. Podczas wywoływania metody `Derived::f()` w używając wskaźnika typu `Base1` lub `Base2` wskaźnik `this` musi zawierać adres początku obiektu klasy `Derived`. Aby otrzymać adres podobiektu klasy `Derived` należy ponownie posłużyć się offsetem. Przesunięcie podobiektu klasy `Base2` w obiekcie innej nieznanej klasy pochodnej (w tym przypadku klasy `Derived`) nie jest znane w czasie kompilacji. Klasa `Base1` nie wie, w którym miejscu dowolnej klasy pochodnej zostanie ułożona. Kompilator musi opóźnić obliczenie tego przesunięcia w czasie wykonania. Niech przesunięcie podobiektu klasy `Base2` względem `Derived` wynosi `offset_Base2`. Wartość `offset_Base2` musi zostać wyliczona podczas wołania metody wirtualnej `f()` niezależnie od tego, w jakim innym obiekcie umieszczony jest w wyniku dziedziczenia wirtualnego podobiekt `Base2`. Popularnym rozwiązaniem jest umieszczenie niezbędnych informacji w zmodyfikowanej tablicy funkcji wirtualnych *tfw*.

Na Rys. 5.8 przedstawiono tablicę funkcji wirtualnych dostosowaną do wołania funkcji wirtualnych w hierarchii klas z dziedziczeniem wielobazowym.



Rys. 5.8 Tablica funkcji wirtualnych dla dziedziczenia wielobazowego

Dla funkcji `Derived::f()` w podobiekcie `Base1` offset wynosi `0`, ponieważ metoda `Derived::f()` przesłania wersję `Base1::f()` i tablica funkcji wirtualnych zawiera poprawny adres. Adresy obiektów `Derived` i `Base1` w ramach obiektu `Derived` są identyczne, dlatego `offset` wynosi zero.

W klasie `Derived` metoda `Base1::g()` nie została przesłonięta, dlatego wpis w tablicy *wtfw* dla metody `g()` w ramach klasy `Derived` zawiera adres `Base1::g()`. Ponieważ podobiekt `Base1` w obiekcie `Derived` i obiekt `Derived` mają identyczne adresy, przesunięcie jest równe zero.

Wpis w polu *offset* dla funkcji `Derived::f()` w podobiekcie `Base2` zawiera informacje pozwalające wykonać polimorficzne wywołanie metody `f()` za pośrednictwem wskaźnika lub referencji typu `Base2`. Ponieważ w klasie `Derived` metoda `Base1::f()` została przesłonięta, wywołanie jest przeniesione do `Derived`. Wskaźnik typu `Base2` zawiera jednak adres podobiektu `Base2` w obiekcie `Derived`, podczas gdy argumentem wywołanie `Derived::f()` musi być adres obiektu `Derived`. Żeby wskaźnik `this` zawierał poprawny adres, od adresu obiektu klasy `Derived` należy odjąć przesunięcie podobiektu `Base2` wewnątrz obiektu `Derived`. Przesunięcie to ma wartość `offset_Base2`.

Implementacja wywołania metody w hierarchii klas z dziedziczeniem wirtualnym

Jeżeli w hierarchii klas dopuszcza się możliwość wielodziedziczenia, możliwe staje się również, że pomiędzy dwoma klasami, bazową i pochodną, istnieją co najmniej dwie ścieżki dziedziczenia. Taką hierarchię zaprezentowano w Przykład 5.6 W opisanej implementacji obiekt wirtualnej klasy bazowej rozmieszczony jest jako ostatni. Wskaźnik `pVirtualBase` jest dodatkowym wskaźnikiem przechowywanym w każdym podobiekcie wykorzystującym klasę `VirtualBase` jako wirtualną klasę bazową. Budowę obiektów klas `VirtualBase`, `_Base_1`, `_Base_2` oraz `Derived` przedstawiono na Rys. 5.9.

Przykład 5.6

Niech dana będzie hierarchia klas.

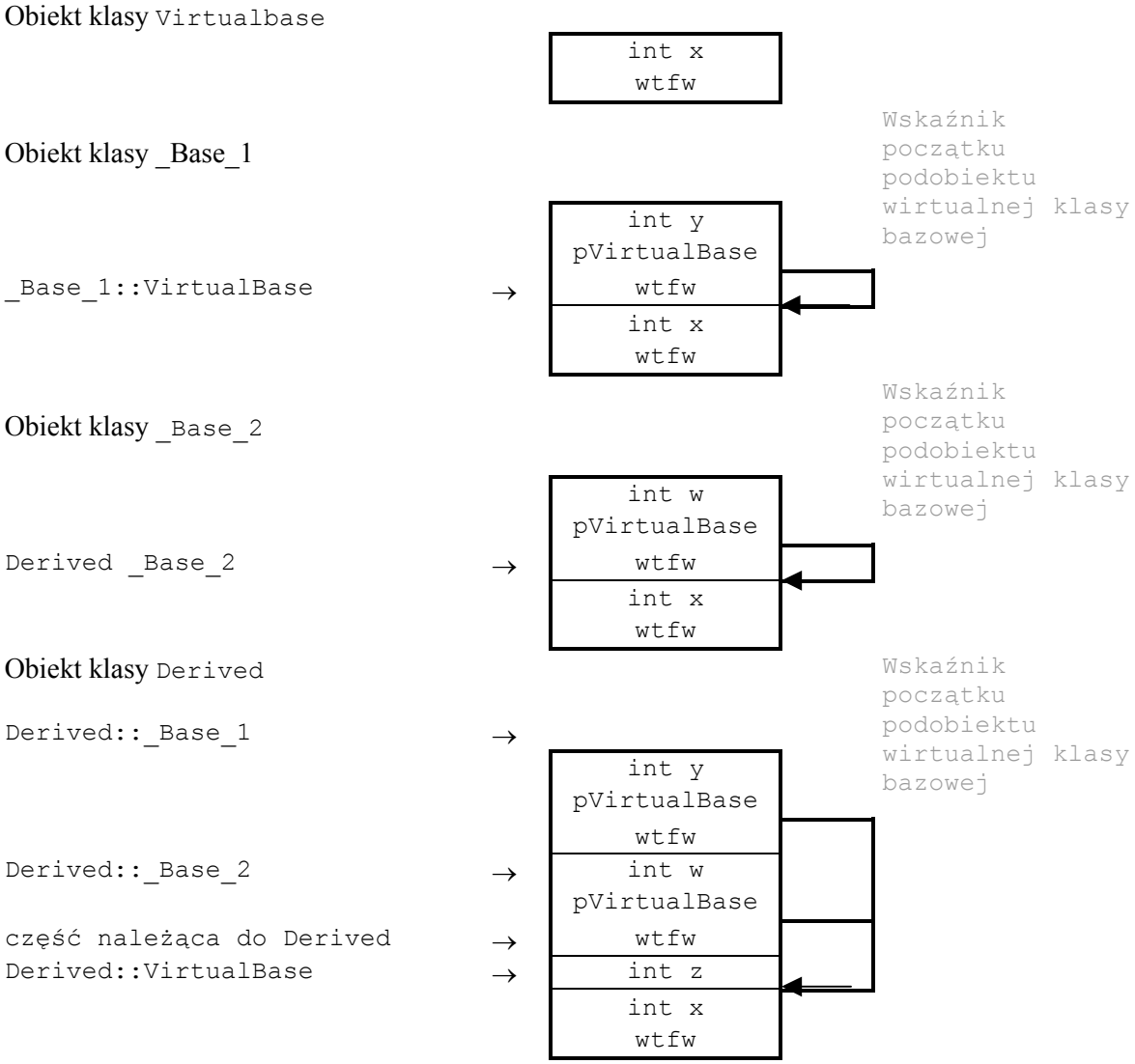
```
class VirtualBase {
public:
5.3..1      virtual void f();
      virtual void g();
      virtual void h();
public:
      int x;
};

class _Base_1 : virtual public VirtualBase {
public:
      virtual void g();
public:
      int y;
};

class _Base_2 : virtual public VirtualBase {
public:
      virtual void f();
public:
      int w;
};

class Derived : public D1, public D2 {
public:
      virtual void h();
public:
      int z;
};
```

Dzięki wirtualnemu dziedziczeniu, klasa `Derived` zawiera tylko jeden podobiekt klasy `VirtualBase` zamiast dwóch, jednego odziedziczonego po `_Base_1` i jednego odziedziczonego po `_Base_2`.
W wyniku tego, odwołując się do pola `VirtualBase::x` w klasie `Derived`, można napisać

```
Derived d;
d.x = 0;
```

i nie spowoduje to błędu kompilacji, gdyż odwołanie nie jest niejednoznaczne. W przypadku braku dziedziczenia wirtualnego, klasa `Derived` zawierała by dwa pola `VirtualBase::x`, jedno w podobiekcie `_Base_1::x` i drugie w podobiekcie `_Base_2::x`. W wyniku tego, aby uniknąć błędów kompilacji, programista musi jawnie wyspecyfikować, które z tych pól ma zostać użyte

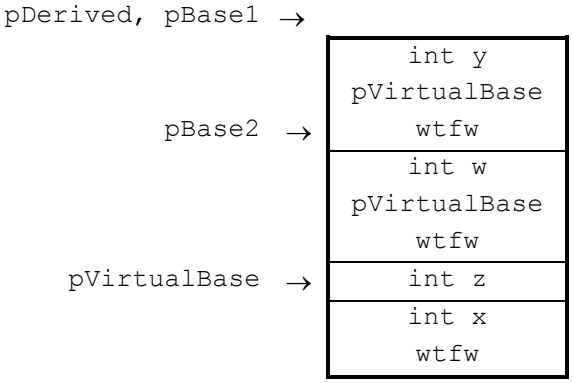
```
// kod jest poprawny, jeżeli pomiędzy klasami VirtualBase,
// _Base_1 oraz _Base_2 nie ma dziedziczenia wirtualnego
Derived d;
d::_Base_1.x = 1;
d::_Base_2.x = 2;
```



Rys. 5.9 Budowa obiektów klas w hierarchii wielobazowej

Przedstawione deklaracje zmiennych `pVirtualBase`, `pBase1` oraz `pBase2` wskazują na różne podobiekty tego samego obiektu klasy `Derived`. Prezentuje to Rys. 5.10.

```
Derived derived;
Derived* pDerived = &derived;
VirtualBase* pVirtualBase = &derived;
_Base_1* pBase1 = &derived;
_Base_2* pBase2 = &derived
```



Rys. 5.10 Wskazania zmiennych wskaźnikowych `pDerived`, `pVirtualBase`, `pBase1` i `pBase2`

6 Architektura komputerów

Jednym z końcowych etapów kompilacji jest generowanie kodu maszynowego. Kompilator może wygenerować kod asemblera a następnie posłużyć się osobnym narzędziem w celu wykonaniu translacji asemblera do kodu maszynowego lub dokonać jej samodzielnie.

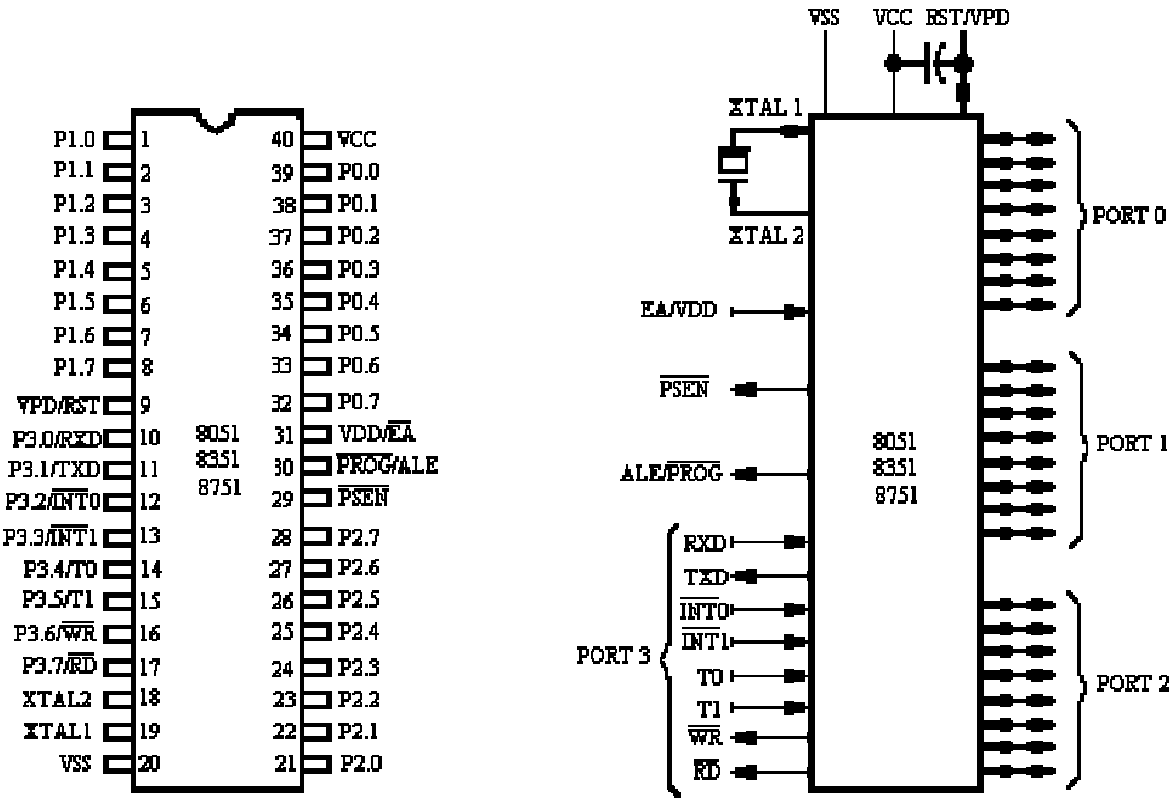
Język asemblera dla danego procesora bardzo ściśle zależy od architektury tego procesora. Ścieżka danych jest sercem każdego komputera. Składa się z pewnej liczby rejestrów (ang. *register*), pewnej liczby magistral (ang. *bus*) i kilku jednostek wykonawczych (ang. *arithmetic – logical unit ALU*). Główna pętla działania to pobranie operandów z rejestrów i przesłanie ich magistralami do jednostek wykonawczych oraz zapisanie wyników operacji w rejestrach.

Ścieżka danych warunkuje takie właściwości komputera jak ilość rejestrów, ilość i szerokość magistral, stosowanie pobierania wstępnego, głębokość potoku, architekturę RISC (*Reduced / Rationalized Instruction Set* Computer) lub CISC (*Complex Instruction Set* Computer) oraz wiele innych.

Chociaż temat pracy jasno wymienia architekturę Intel x86 jako docelową dla generowanego kodu asemblera, w celu nadania pracy bardziej badawczego charakteru, do analizy przyjęto trzy architektury procesorów 8051, Intel Pentium 4 oraz UltraSPARC III. Należy również zaznaczyć, że kompilator dostarczony do pracy generuje kod tylko dla wspomnianej w temacie architektury Intel x86.
Procesor 8051 jako przedstawiciel prostego układu stosowanego w systemach wbudowanych (ang. *embedded* systems), Intel Pentium 4 jako układu do komputerów osobistych oraz UltraSPARC III Cu firmy Sun jako układ używany w wysokowydajnych serwerach wieloprocesorowych. Procesor 8051 ma jedną magistralę główną z przłączonymi rejestrami i jedną jednostką ALU.
W Pentium 4 stosuje się skomplikowany mechanizm konwersji złożonych rozkazów o charakterze CISC na mikrooperacje RISC oraz przekazaniu ich do superskalarnego rdzenia RISCowego, gdzie mogę być wykonane w zmienionej co do pierwotnej kolejności.
Procesor UltraSPARC III Cu jest prostszy w budowie od układu Intela. Jest on przykładem architektury RISC, w której instrukcje poziomu ISA (ang. *ISA - Instruction Set Architecture*), czyli instrukcje asemblera są już mikrooperacjami. W ten sposób, omija się translację złożonych instrukcji na mikrooperacje. Inicjowanie, wykonywanie oraz zwalnianie rozkazów odbywa się tutaj zawsze zgodnie z ich kolejnością w programie, ale w jednym cyklu zegara można rozpocząć więcej niż jedną instrukcję.

6.1 Procesor 8051

Procesor 8051 jest prostym 8 bitowym procesorem. Składa się on z około 60 000 tranzystorów. Od początku nie był projektowany jako układ efektywny pod względem mocy obliczeniowej tylko jako układ tani. Procesor ten jest mikrokontrolerem używanym głównie w urządzeniach wbudowanych. Schemat jego wyprowadzeń zaprezentowano na Rys. 6.1.



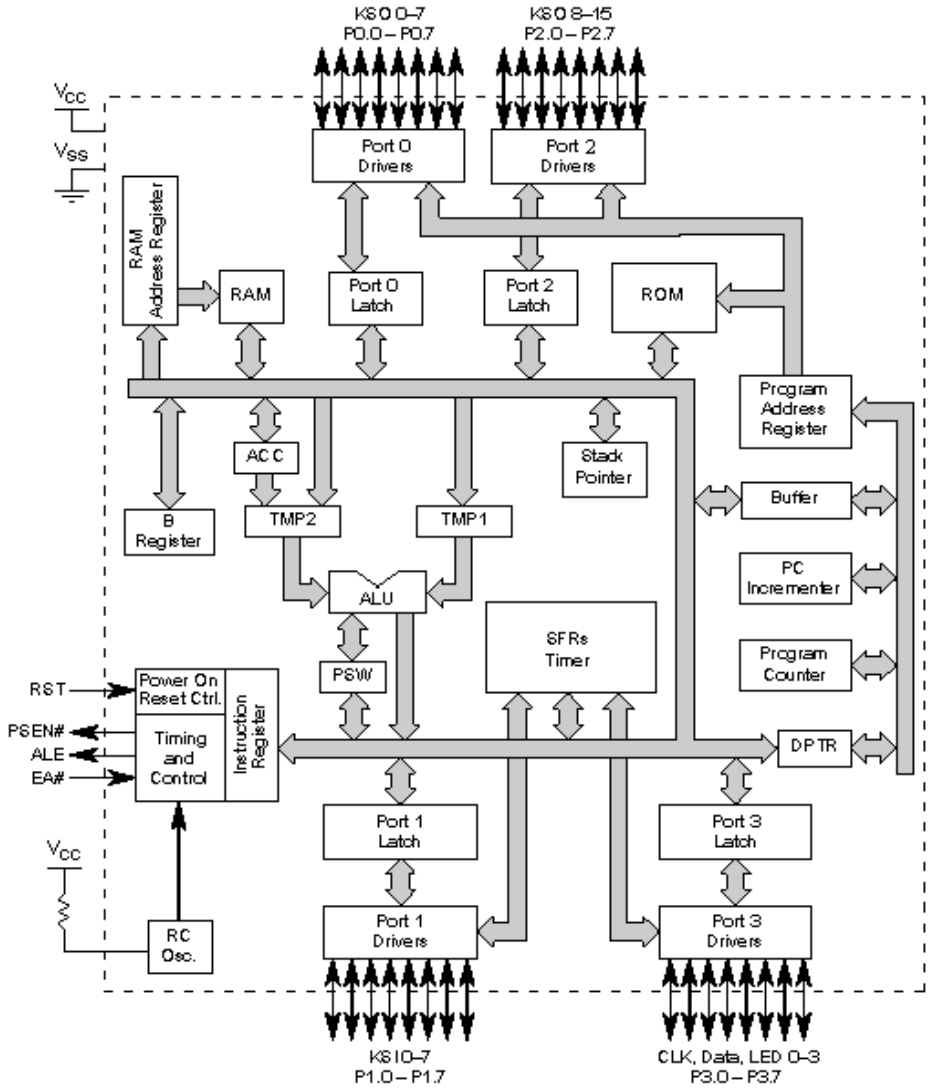
Rys. 6.1 Fizyczny oraz logiczny rozkład wyprowadzeń procesora 8051 [21]

Układ umieszczono w 40 stykowej obudowie. 16 linii to linie adresowe, dzięki czemu procesor może zaadresować 64KB pamięci. Magistrala danych ma szerokość 8 bitów i posiada 32 linie wejścia-wyjścia zorganizowane w cztery grupy po 8 bitów. Do każdej z tych linii można podłączyć urządzenie zewnętrzne.

Żeby zmniejszyć liczbę wyprowadzeń, osiem najmniej znaczących linii adresowych to linie multipleksowane z liniami danych. W trakcie transakcji magistralowej linie te służą do przekazania adresu w pierwszym cyklu i do przesyłania danych w kolejnych cyklach. Opis sygnałów wraz z podaniem ich znaczenia przedstawiono w Tabela 19.

Tabela 19 Sygnały procesora 8051

LINIA SYGNAŁU	OPIS SYGNAŁU
WR# RD#	Gdy procesor korzysta z pamięci zewnętrznej, linie WR# (ang. *write*) i RD# (ang. *read*) określają rodzaj żądania, odpowiednio odczyt lub zapis.
PSEN#	(ang. *Program Store Enable*) – sygnalizuje odczyt z pamięci programu.
EA#	Jeżeli sygnał ma wartość logiczną 1, sygnalizuje, że wewnętrzna 4-kilobajtowa pamięć (8KB w 8052) układu jest przypisana do dolnego zakresu przestrzeni adresowej, a pamięć zewnętrzna do adresów wyższych. Jeżeli sygnał ma wartość logiczną 0 pamięć zewnętrzna wykorzystuje całą przestrzeń adresową, a pamięć wewnętrzna nie jest używana.
T0 T1	(ang. *timer*) – dwie linie zegarowe umożliwiają przyłączenie do procesora zewnętrznych układów zegarowych
I0 I1	(ang. *interrupt*) – dwie linie przerwań umożliwiają przyłączanie do procesora zewnętrznych układów zegarowych.
TXD RXD	Linie umożliwiają szeregową komunikację z terminalem lub modemem.
RST	(ang. *reset*) – linia umożliwia na zerowanie procesora



Rys. 6.2 Schemat architektury procesora 8051 [29]

Sercem procesora jest jego magistrala do której podłączone są rejestry. Rejestr procesora jest komórką pamięci zbudowaną w chip o bardzo krótkim czasie dostępu. Opis rejestrów zamieszczono w Tabela 20.

Tabela 20 Rejestry procesora 8051

SYMBOL REJESTRU	NAZWA REJESTRU	ROZMIAR (w bitach)	ZASTOSOWANIE
ACC	*acumulator*	8	główny rejestr arytmetyczny, służy do przechowywania argumentów operacji matematycznych jak również ich wyników
B		8	rejestr arytmetyczny
SP	*stack pointer*	8	wskaźnik stosu
IR	*instruction register*	8	przechowuje aktualnie wykonywany rozkaz
PC	*program counter*	16	przechowuje adres kolejnego rozkazu do wykonania, nie ma dostępu do magistrali głównej
PC Inc.	*pc incrementer*	8	*pseudorejestr*, wpisanie do niego wartości adresu i jego odczytanie powoduje automatyczne zwiększenie adresu o jeden, nie ma dostępu do magistrali
TMP1 TMP2		8	operandy operacji arytmetycznych kopiowane są do tych rejestrów, skąd następnie pobierane są przez jednostkę arytmetyczno – logiczną ALU
PSW	*program status word*	8	rejestr specjalny, zwany *słowem stanu programu* C A RS O P C – *carry*, przepełnienie najstarszego bitu A – *auxiliary carry*, przeniesienie z młodszej do starszej tetrady RS – bity numeru zbioru rejestrów O – *overflow*, przepełnienie w kodzie U2 P – *parity*, bit parzystości pozostałe pola nie są dostępne dla użytkownika
RAM	*address register*	8	rejestr do adresowania pamięci RAM na dane programu
ROM	*data register*	16	rejestr adresuje pamięć ROM na kod programu
DPTR	*double pointer*	16	rejestr notatnikowy do zarządzania i przechowywania 16 bitowych adresów
BUFFER	*buffer*	16	rejestr notatnikowy
IE	*interrupt enable*	8	rejestr specjalny, pozwala na blokowanie przerwań EA E2 ES E1 X1 E0 X0 EA – jeżeli bit ma wartość 0 wszystkie przerwania są blokowane niezależnie od pozostałych bitów E2, E1, E0 – związane z trzema kanałami zegarowymi, od których przerwania można obsługiwać osobno ES – wartość bitu równa zero bokuje przerwania z kanału komunikacji szeregowej pozostałe bity sterują przerwaniami z urządzeń zewnętrznych

Tabela 20 Rejestry procesora 8051

SYMBOL REJESTRU	NAZWA REJESTRU	ROZMIAR (w bitach)	ZASTOSOWANIE
IP	*interrupt priority*	8	rejestr specjalny, określa poziom priorytetu przerwań, przerwania o niskim priorytecie mogą być przerwane przez przerwania o wyższym priorytecie | | | E2 | ES | E1 | X1 | E0 | X0 | ustawienie bitu oznacza wysoki priorytet przerwania
TCON	*timer control*	8	steruje głównymi układami czasowo – licznikowymi. | O1 | R1 | O0 | R0 | E1 | T1 | E0 | T0 | O1, O0 – bity są ustawiane sprzętowo w przypadku przepełnienia licznika R1, R0 – bity te sterują uruchomieniem zegarów pozostałe sterują sposobem wyzwalania zegarów, zboczem lub poziomem sygnału
TMOD	*timer mode*	8	rejestr specjalny, decyduje o sposobie pracy zegarów, można ustawić ośmio, trzynasto lub szesnastobitowy tryb pracy, jak również charakter pracy jako zegar lub licznik | zegar 1 | zegar 2 |

Układ 8051 wykorzystuje dwie osobne pamięci. Jedna tylko do odczytu ROM o pojemności do 64KB przeznaczona jest na kod programu, druga z możliwością pisania o pojemności 128 bajtów (256 bajtów dla układu 8052) przechowuje dane programu. Procesor 8051 zawiera cztery 8-bitowe porty wejścia-wyjścia (*Port 1 Diverse, Port 2 Diverse, Port 3 Diverse, Port 4 Diverse*), umożliwiające sterowanie do 32 zewnętrznych urządzeń.

Układ 8051 jest układem synchronicznym, w którym większość rozkazów trwa jeden cykl zegara *SFRe Timer*. Cykl zegara można podzielić na sześć stanów.

1. Rozkaz jest pobierany z pamięci ROM, umieszczany na głównej magistrali i przesyłany do rejestru IR.
2. Rozkaz jest dekodowany, wartość rejestru PC jest inkrementowana.
3. Pobierane są operandy.
4. Jeden operand przesyłany jest do rejestru TMP1. Opcjonalnie można drugi operand skopiować z rejestru ACC do rejestru TMP2.
5. Jednostka ALU wykonuje operacje na operandach.

Wynik zwrócony przez ALU jest zapisywany, rejestr ROM ADDR przygotowywany jest do pobrania kolejnego rozkazu.

Procesor 8051 zaprojektowany został do wykonywania tylko jednego zadania, dlatego nie wspiera mechanizmów wielozadaniowości i nie posiada jakichkolwiek mechanizmów ochrony pamięci.
Procesor w celu bardzo szybkiej obsługi przerwań na cztery zestawy rejestrów. W danej chwili dostępny jest tylko jeden zestaw, wskazywany przez dwubitowe pole rejestru PSW. W momencie wystąpienia zewnętrznego przerwania, procesor nie musi zachowywać wartości rejestrów, wystarczy, że przełączy aktualny zestaw. Dzięki temu praca w systemach rzeczywistych, gdzie przerwania zewnętrzne mogą nadchodzić z bardzo dużą częstotliwością mogą być wydajnie obsługiwane.

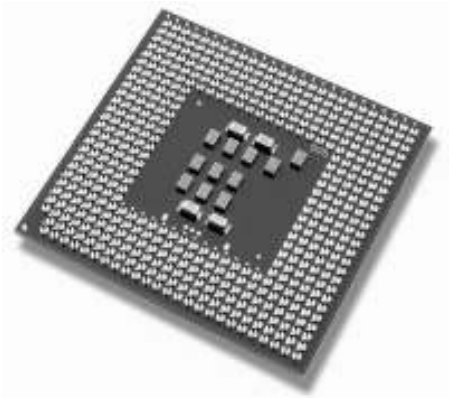
Rejestry procesora są skojarzone w początkowymi adresami pamięci. Rejestr R0 z zestawu nr 0 jest równoważny z 0 bajtem w pamięci. Zapisując wartość w rejestrze R0 można ją odczytać z pamięci z bajtu o adresie 0. Kolejne zestawy rejestrów odpowiadają kolejnym bajtom pamięci. Pamięć potrzebna na odwzorowanie 4 zestawów ośmiu ośmiobitowych rejestrów wynosi 256 bitów, czyli 32 bajty. Po 32 początkowych bajtach pamięci następuje16 bajtów pamięci adresowanej bitowo. 16 bajtów pamięci adresowanej bitowo umożliwia odwołanie się do 128 równych bitów numerowanych od 0 do 127. Operowanie na bitach jest bardzo często konieczne w związku ze sterowaniem binarnym stanem rozmaitych przełączników, diod itp., które można podłączyć do procesora. Rejestry specjalne PSW, IE, IP, TCON, TMOD skojarzone są z pamięcią od bajtu 128 do 255.

6.2 Intel Pentium 4

Procesor Intel Pentium 4 powstał w listopadzie 2000 roku. Jest on procesorem zbudowanym z 42 milionów tranzystorów wykonanym w technologii 0.18µ taktowany zegarem 1.5GHz. W ciągu kilku kolejnych lat firma Intel zwiększyła częstotliwość zegara do 3.2GHz a ilość tranzystorów na chipsecie wzrosła do 55 milionów.

Pomimo, że ilość tranzystorów wzrosła prawie 2000 razy w porównaniu do procesora 8086 Intel Pentium 4 jest w pełni kompatybilny wstecz i może wykonać każdy kod napisany na dowolny procesor w linii x86. Od strony sprzętowej Pentium 4 jest procesorem 64-bitowym, chociaż dla programisty 64-bitowy tryb do pamięci nie jest dostępny a procesor zachowuje się jak układ 32-bitowy.

Obudowę procesora zaprezentowano na Rys. 6.3. Układ na kształt kwadratu o długości boku równej 35mm. Na spodniej części umieszczono 478 wyprowadzeń w kształcie macierzy o wymiarach 26 x 26 z pominięciem kwadratowej przestrzeni w środku o wymiarach 14 x 14. W jednym z narożników producent nie umieścił dwóch wyprowadzeń w celu uniemożliwienia błędnego umieszczenie procesora w gnieździe. W pośród 478 wyprowadzeń, 85 doprowadza zasilanie a 180 jest połączonych z masą w celu niwelowania zakłóceń oraz 15 końcówek nieużywanych możliwych do wykorzystania w przyszłości. Logiczny schemat wyprowadzeń procesora Intel Pentium 4 pokazano na Rys. 6.3.



Rys. 6.3 Obudowa procesora Intel Pentium 4 [21]

Sygnały zostały pogrupowane w logiczne grupy zw zależności od pełnionej funkcji. Opis grup, ich roli w systemie a także wchodzących w ich skład sygnałów zaprezentowano w Tabela 21.

Tabela 21 Opis sygnałów procesora Intel Pentium 4

NAZWA GRUPY	OPIS	SYGNAŁY	OPIS
Arbitraż magistrali	obsługuje żądania dostępu do magistrali czyli *arbitraż*	BR0	Sygnałem tym urządzenie sygnalizuje żądanie dostępu do magistrali
		BRPI#[2]	Urządzenie sygnalizuje żądanie o wysokim priorytecie
		LOCK#	Procesor może tym sygnałem zablokować magistralę
Żądanie	po przejęciu kontroli urządzenie w tym również sam procesor mogą sformułować właściwe żądanie	A#	(ang. *address*), adresy wysyłane na magistralę są 36 bitowe, ale ich trzy najmniej znaczące bity są zawsze równe zero dlatego nie są im przypisywane żadne linie i w celu przesłania adresu wystarczają tylko 33 bity
		ASD#	Sygnał jest włączany gdy linie adresowe są włączone, informuje odbiorcę, że stan linii adresowych jest poprawny
		REQ#	(ang. *request*), sygnał określa typ pracy magistrali
		Parity#	Dwie linie zabezpieczają poprawność danych dwóch sygnałów A# i REQ#
Błąd	linie mają charakter diagnostyczny	Misc	Linie służą do zgłaszania błędów arytmetyki zmiennoprzecinkowej, wewnętrznych i innych
Odpowiedź	obejmuje sygnały wykorzystywane przez urządzenia podrzędne	RS#	Kod stanu
		TRDY#	Urządzenie podrzędne tym sygnałem informuje o swojej gotowości do odbierania danych
		Parity#	Kontrola parzystości
		BNR#	Urządzenie podrzędne sygnalizuje, że jest zajęte i nie może odpowiedzieć wystarczająco szybko
Dane	linie służą do przesyłania danych	D#	Sygnał ten powoduje umieszczenie na magistrali 8 bitów danych.
		DRDY#	Procesor włącza sygnał po umieszczeniu na magistrali 8 bitowej porcji danych (patrz sygnał D#).
		DBSY#	Linia informuje wszystkie urządzenia o zajętości magistrali
		Parity#	Kontrola parzystości
		Misc	Linie służą do sygnalizowania błędów
RESET#	Sygnał resetuje procesor na skutek awarii lub restartu systemu		
Zarządzanie zasilaniem	Procesor może być zasilany jednym z kilku napięć, ale musi wiedzieć którym		
Zarządzanie termiczne	Zapobiega przez przegrzaniem procesora		
Diagnostyka	Sygnały do testowania i debugowania systemu zgodnie z normą IEEE 1149.1 JTAG		
Inicjalizacja	Linie używane przy starcie (ang. *boot*) systemu		

[2] Zapis SIGNAL# oznacza, ze sygnał procesora SIGNAL aktywowany jest stanem niskim

Rejestry procesora Intel Pentium 4

Rejestry procesora Intel Pentium zaprezentowano w Tabela 22. Można je podzielić na cztery grupy: rejestrów uniwersalnych EAX, EBX, ECX, EDX, rejestrów wskaźnikowych i uniwersalnych ESI, EDI, EBP, ESP, rejestrów segmentowych CS, SS, DS., ES, FS, GS oraz wskaźnik poleceń EIP i rejestr flag EFLAGS.

Tabela 22 Rejestry procesora Intel Pentium 4

NAZWA	ROZMIAR W BITACH	OPIS
EAX	32	*Acumulator* – rejestr wykorzystywany w operacjach arytmetycznych, logicznych oraz przesyłania danych.
EBX	32	Może wskazywać położenie, lokalizację w pamięci. Używany jako podstawa do wyznaczania adresu. Nazywany jako wskaźnik ramki (ang. *frame pointer*).
ECX	32	*Counter* – używa się go głównie jak licznika w powtarzających się fragmentach kodu.
EDX	32	Jest używany głównie jako wskaźnik adresów w rozkazach IN i OUT. Rejestr jest używany również w operacji dzielenia.
ESI	32	*Source Indicator* – używany jako wskaźnik pamięci jak również wskazuje miejsce skąd są przesyłane dane.
EDI	32	*Destination Indicator* – wskazuje miejsce dokąd są przesyłane dane.
EBP	32	Podobnie jak (E)BX, (E)SI, (E)DI może być użyty jako wskaźnik pamięci z pewną różnicą, że odnosi się do segmentu (E)SS a nie do (E)BP
ESP	32	(E)SP Stack Pointer – podaje położenie bieżącego wierzchołka stosu.
CS SS DS ES FS GS	16 16 16 16 16 16	Rejestry segmentowe, w procesorze 8088 służyły do adresowania 2^{20} bajtów pamęci przy użyciu 16-bitowych adresów. CS – *code segment* – wskazuje segment kodu, SS – *stack segment* – wskazuje segment stosu, DS – *data segment* – wskazuje segment danych, ES, FS, GS rejestry segmentowe dodatkowe,
EIP	32	*Extended Instruction Pointer* – licznik rozkazów, zawiera offset pamięci w którym zawarty jest następny rozkaz do wykonania. Bazowy adres mieści się w rejestrze CS.
EFLAGS	32	Rejestr flag jest 32 bitowym rejestrem dającym programiście informacje o szczegółowych informacjach na temat procesora. Składa się z bitów: CF – *carry flag* – flaga przeniesienia, ma wartość 1 jeżeli nastąpiło przeniesienie z najstarszego bitu PF – *parity flag* – znacznik parzystości AF – *auxiliary flag* – znacznik przeniesienia pomocniczego (1 jeżeli nastąpiło przeniesienie z 3 na 4 bit ZF – *zero flag* – znacznik zera, 1 jeżeli wynikiem obliczeń jest zero SF – *sign flag* – znacznik znaku, 1 jeżeli otrzymano liczbę ujemną TF – *trap flag* – znacznik pracy krokowej umożliwiającej debugowanie IF – *interrupt flag* – znacznik zezwolenia na przerwanie, określa

Tabela 22 Rejestry procesora Intel Pentium 4

NAZWA	ROZMIAR W BITACH	OPIS
		czy przerwanie będzie wykonane natychmiast po wystąpieniu OF – *overflow flag* – znacznik przepełniania, flaga ustawiana jeżeli wynik jest tak dużą liczbą, że nie można jej zapisać na 32 bitach IOPL – *I/O privilege level* – rejestr dwubitowy, określa priorytet procesu NT – *nested task flag* – znacznik zagnieżdżenia, bit ma wartość 1 jeżeli proces wywołał inny proces RF – *resume flag* – znacznik wznowienia, używany przy debugowaniu w celu wznowienia pracy po złapaniu wyjątku VM – *virtual mode flag* – określa tryb pracy procesora, wartość 1 jako tryb wirtualny, 0 jako jeden z pozostałych: rzeczywisty, chroniony, emulacji 286 AC – *alignment check* – określa czy ma wystąpić błąd jeżeli procesor pracuje na niewyrównanych danych VIF – *virtual interrupt flag* – bit określa sposób obsługi przerwań wirtualnych ID – *identyfication* – umożliwia odczyt wersji procesora

Procesor Intel Pentium 4 oprócz rejestrów procesora zawiera także zbiór rejestrów koprocesora. Są one zorganizowane w postaci stosu i można je pogrupować w następujący sposób.

- osiem adresowalnych 80-bitowych rejestrów numerycznych, zorganizowanych jako stos. Rejestr ten zbudowany jest z trzech części. Część mantysy ma 64 bity szerokości, część wykładnika 15 bitów szerokości, część znaku o szerokości jednego bitu.
- trzy 16-bitowe rejestry zawierające słowo stanu koprocesora, słowo sterowania koprocesora oraz słowo stanu zawartości rejestrów stosu koprocesora
- rejestry służące do wskazywania rozkazu (rejestr wskaźnika rozkazu) oraz argumentu (rejestr wskaźnika argumenty)
- 11-bitowy rejestr zawierające kod operacyjny (ang. *opcode*)

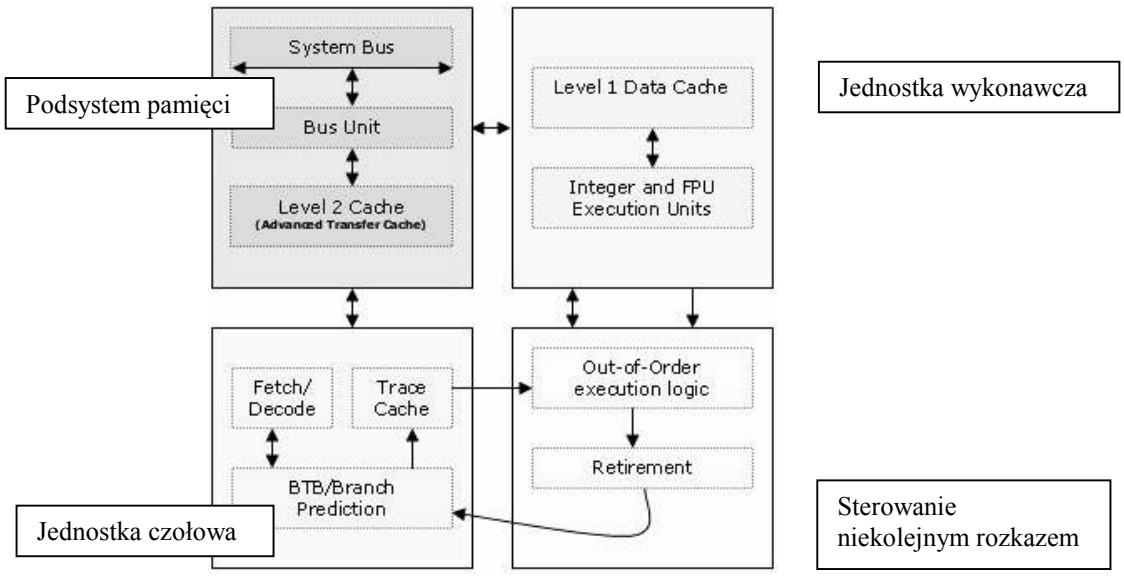
Rejestry numeryczne zorganizowane są w postaci stosu i nazwane są ST(0), ST(1), … ST(7). Na wierzchołku stosu znajduje się rejestr ST(0) nazywany w skrócie ST i pełni on funkcję akumulatora.

Z punktu widzenia programisty ważną rzeczą programując w procesorze Intel Pentium 4 są jego tryby pracy. Tryb pracy, w którym aktualnie znajduje się procesor opisywany jest bitem VM z rejestru EFLAGS.

Architektura NetBurst

Układ Pentium 4 składa się z czterech podstawowych części, podsystemu pamięci, jednostki czołowej, układu sterowania niekolejnym wykonywaniem rozkazów i jednostek wykonawczych.
Podsystem pamięci zawiera jednolitą pamięć podręczną drugiego poziomu L2 oraz układy dostępu do pamięci zewnętrznej RAM. Pamięć podręczna zawiera układ wstępnego pobierania, który próbuje pobrać dane z pamięci zanim jeszcze inne układy procesora wyślą żądanie dostępu do nich.
Zadaniem jednostki czołowej jest pobieranie rozkazów z pamięci L2 i dekodowanie ich z zachowaniem kolejności w programie. Rozkazy złożone są również rozbijane na RISCowe mikroinstrukcje. Dla zwiększenia efektywności translacji złożonych CISCowych rozkazów na odpowiadające im RISCowe sekwencje mikrooperacji w jednostkę czołową zbudowana jest pamięć ROM zawierające gotowe algorytmy translacji. Przekształcone instrukcje przekazywane są do pamięci podręcznej śledzenia (ang. *execution trace cache*). Dzięki buforowaniu zdekodowanych mikrooperacji, nie ma potrzeby ich ponownego dekodowania przy kolejnym wykonaniu.
Rozkazy z pamięci śledzenia są przekazywane do jednostki układu szeregowania w kolejności, w jakiej zostały umieszczone w programie. Jednostka szeregowania decyduje o kolejności, w jakiej rozkazy będą wykonane w jednostce wykonawczej. Rozkazy, których nie można jeszcze wykonać, wstrzymywane są przez planistę. Planista jednocześnie dąży do tego, żeby nie przerywać strumieniowego przetwarzania rozkazów i stara się zainicjować kolejną instrukcję. Pomimo, że inicjowanie rozkazów może następować w kolejności różnej od kolejności, w jakiej były ułożone w pamięci śledzenia, mechanizm przerwań wbudowany w układ jednostki zwalniania (ang *retirement*) zapewnia kończenie wykonywania rozkazów w naturalnej kolejności.
Procesor Pentium 4 zawiera kilka zestawów jednostek wykonawczych dedykowanych instrukcjom stało i zmiennopozycyjnych. Mogą współpracować równolegle zwiększając moc obliczeniową procesora. Korzystają z danych zapisanych w rejestrach w pamięci podręcznej L1.
W najnowszych modelach procesorów Intel architektura NetBurst została wyparta przez technologie dublowania rdzeni procesora zwaną jako Intel Core.

Schemat architektury NetBurst pokazano na Rys. 6.4.

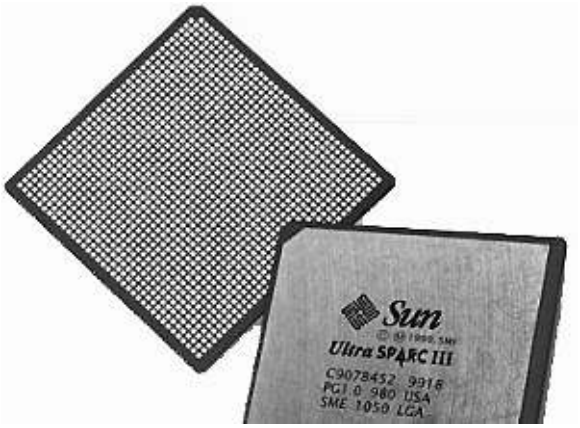


Rys. 6.4 Elementy architektury NetBurst [21]

6.3 UltraSPARC III Cu

Procesor UltraSPARC III Cu[3] firmy Sun[4] jest klasycznym przykładem architektury RISC (ang. *rationalized (reduced) instruction set computers*). Lista instrukcji tego procesora jest o wiele krótsza od listy procesora Intela a same instrukcje są o wiele prostsze. Ponadto wiele instrukcji syntetyzuje się przy pomocy innych. Dzięki temu można opuścić jeden krok translacji złożonych instrukcji asemblera na mikrooperacje. Podstawowe rozkazy same w sobie już są mikrooperacjami.. Chociaż procesor ten może być używany w stacjach roboczych, zaprojektowany został z myślą o wykorzystaniu go w wieloprocesorowych wydajnych serwerach. Układ ten jest w pełni 64-bitowy o 64-bitowych rejestrach i 64-bitowej ścieżce danych. W celu zapewniono kompatybilności wstecz z 32-bitowymi wersjami procesora, procesor umożliwia wykonywanie 32-bitowych programów bez żadnych modyfikacji.

Procesor UltraSPARC III Cu jest osadzony w obudowie typu Land Grid Array (matryca pól kontaktowych) o 1368 wyprowadzeniach ułożonych w macierz 37 x 37. W celu zabezpieczenia przed niewłaściwym umieszczeniem procesora na podstawce jedna z końcówek została usunięta. Procesor ten przedstawiony jest na Rys. 6.5.

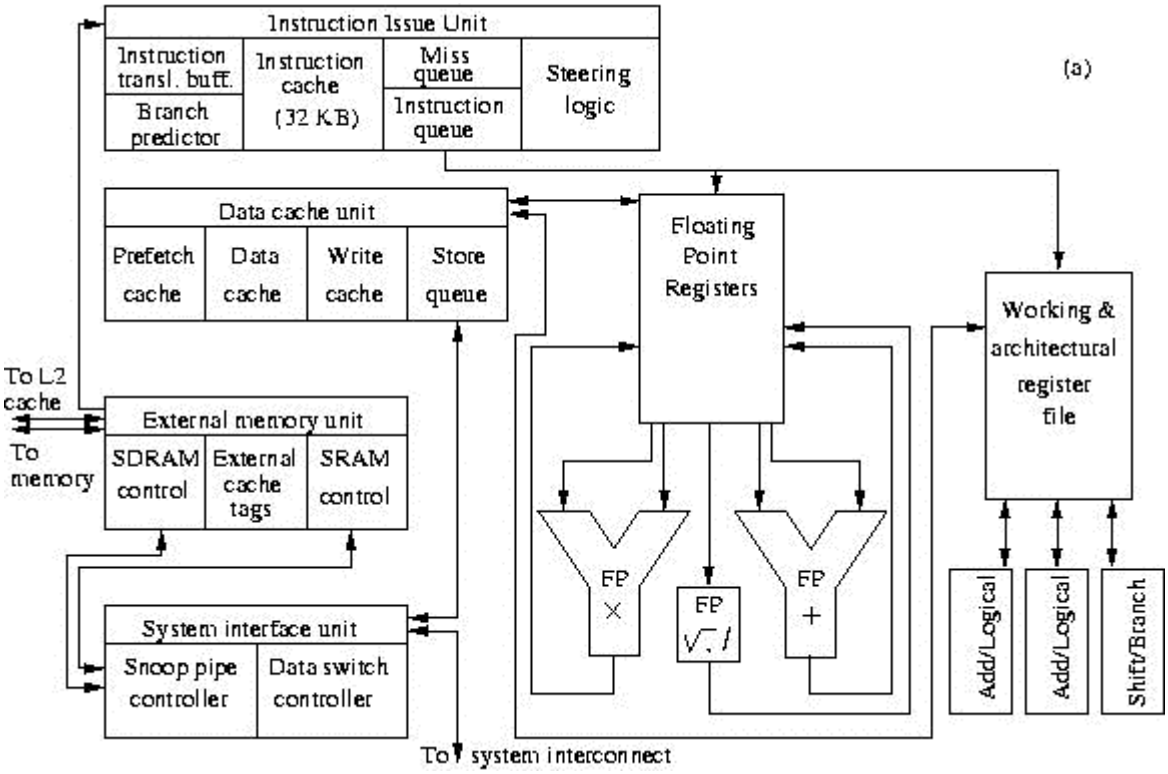


Rys. 6.5 Obudowa procesora Ultra SPARC III Cu [21]

Procesor ma dwie wewnętrzne pamięci podręczne pierwszego poziomu o rozmiarze 32KB na rozkazy i 64KB na dane. Pamięć drugiego poziomu L2 nie jest wbudowana w chipset, lecz jest podłączana jako zewnętrzny układ.
Procesor ma 43-bitową szynę adresową, co umożliwia adresowanie 8TB pamięci, natomiast szyna danych ma szerokość 128 bitów. Jak już zostało wspomniane, układ Suna został zaprojektowany do wykorzystania w wieloprocesorowych serwerach. Procesor wspiera system łączenia wielu procesorów i pamięci implementując mechanizm Ultra Port Architecture UPA.
Układ Suna posiada 32-kilobajtową, czterokanałową pamięć asocjacyjną rozkazów o rozmiarze wiersza 32 bajty. Ponieważ większość rozkazów ma 4 bajty długości, w pamięci tej można przechowywać około 8 tysięcy rozkazów.
Schemat procesora UltraSPARC III Cu jest przedstawiony na Rys. 6.6. Układ ten jest znacznie prostszy w budowie niż układy Intela, ponieważ realizuje on znacznie prostszy poziom maszynowy.

[3] Skrót *Cu* pochodzi od symbolu pierwiastka chemicznego miedzi, z którego zostały wykonane ścieżki procesora, w odróżnieniu od poprzednich modeli w których używano aluminium.
[4] Firma Sun projektuje układy, ich fizycznym wytwarzaniem zajmuje się firma Texas Instruments.

Procesor UltraSPARC III Cu ma dwie grupy rejestrów. Pierwszą z nich tworzę 32 64-bitowe rejestry uniwersalne, drugą 32 rejestry zmiennopozycyjne. Rejestry uniwersalne mają oznaczenia od R0 do R31. Nazwy i funkcje wspomnianych rejestrów zostały zestawione w Tabela 23.



Rys. 6.6 Schemat budowy procesora Ultra SPARC III Cu [21]

Tabela 23 Rejestry procesora UltraSPARC III Cu

REJESTR	NAZWA ALTERNATYWNA	FUNKCJA
R0	G0	Zawiera permanentnie wartość 0. Próby zmiany tej wartości nie są traktowane jako błąd, są jednak zupełnie ignorowane
R1 – R7	G1 – G7	Przechowują wartości zmiennych globalnych
R8 – R13	O0 - O5	Przekazują wartości zmiennych globalnych
R14	SP	Wskaźnik stosu
R15	O7	Rejestr roboczy
R16 – R23	L0 – L7	Przechowują wartości zmiennych lokalnych bieżącej procedury
R24 – R29	I0 – I5	Przechowują parametry wywołania bieżącej procedury
R30	FP	Wskaźnik podstawy bieżącej ramki stosu
R31	I7	Adres powrotu z bieżącej procedury

Przyporządkowanie różnych funkcji różnym rejestrom nie wynika ze sposobu implementowania rejestrów na poziomie sprzętowym, lecz jest w dużej mierze sprawą przyjętej konwencji. Zbiór 32 rejestrów ogólnego przeznaczenia nosi nazwę okna rejestrów (ang. *register windows*). Istnieje wiele zestawów rejestrów uniwersalnych a w danej chwili widoczny jest tylko jeden wskazywany przez rejestr CWP (ang. *current window pointer*). Zadaniem okien rejestrów jest emulacja stosu rejestrów, dzięki czemu przełączanie kontekstu i zapamiętanie wartości rejestrów może być bardzo szybkie a sam

mechanizm wywołań procedur bardzo efektywny. Rozkaz wywołania procedury ukrywa wykorzystywany zestaw rejestrów i udostępnia wywołanej procedurze nowy poprzez zmniejszenie wartości CWP. Niektóre wartości rejestrów są przenoszone z procedury wywołującej do procedury wywoływanej – w ten sposób bardzo wydajnie przekazywane są parametry. Metoda ta opiera się na przemianowywaniu rejestrów. Po wywołaniu procedury wcześniejsze rejestry wyjściowe numerowane od R8 do R15 są w dalszym ciągu widoczne, ale są już rejestrami wejściowymi, czyli numerowanymi od R24 do R31. Zestaw rejestrów globalnych pozostaje bez zmian. W głębokich zagnieżdżeniach procedur może wystąpić sytuacja, że zabraknie okien rejestrów. W takiej sytuacji, zawartość najdawniejszego okna jest kopiowana do pamięci, a okno może być wykorzystane ponownie. Procesor UltraSPARC III Cu ma też 32 rejestry zmiennopozycyjne, które mogą przechowywać liczby zmiennopozycyjne pojedynczej precyzji (32-bitowe) lub podwójnej precyzji (64-bitowe). Możliwe jest też operowanie na liczbach zmiennopozycyjnych poczwórnej precyzji, z których każda zapisywana jest w parze rejestrów zmiennopozycyjnych.

Dwa rejestry wyznaczają bieżącą ramkę. FP, *frame pointer*, wskazuje jej początek i służy do adresowania zmiennych lokalnych. SP*, stack pointer*, wskazuje bieżący wierzchołek stosu i zmienia swoją wartość przy okazji operacji stosowych `push` i `pop`.
Jednostka inicjowania rozkazów (ang. *instruction issue unit*) przygotowuje w każdym cyklu zegara do czterech rozkazów. W przeciwieństwie do procesorów CISC Intela, które taktowane są o wiele szybszym zegarem, procesor UltraSPARC III Cu w każdym cyklu zegara może zainicjować wykonywanie do czterech instrukcji a nie jednej. Dzięki temu 1.2GHz procesor Suna może być podobnie wydajny jak 4.8GHz układ Intela. Mniejsza ilość przygotowanych rozkazów niż cztery może wynikać z chybień w pamięci podręcznej i potrzeby pobrania instrukcji z pamięci operacyjnej. W przypadku wykonywania skoków warunkowych, układ podejmuje decyzje o wyborze następnego rozkazu. Decyzja czy skok będzie wykonany prognozowana jest na podstawie tablicy rozgałęzień (ang. *branch table*), która jest w stanie pomieścić 16K wpisów. Aby zapewnić ciągłość potoku, prognozowane rozkazy kopiowane są do bufora.

Wyjście bufora rozkazów połączone jest z jednostkami wykonawczymi. Jednostki te są wyspecjalizowane jako stało oraz zmiennopozycyjne, a także dostępu do pamięci. Jednostka stałopozycyjna zawiera dwa układy ALU oraz krótki potok rozkazów rozgałęzień z własnym zestawem rejestrów. Jednostka zmiennopozycyjna zawiera 32 rejestry i trzy niezależne układy ALU dodawania – odejmowania, mnożenia i dzielenia. Odpowiada ona również za wykonanie instrukcji graficznych. Jednostka ładowania i zapamiętywania odpowiada za rozkazy odczytu i zapisu danych. Jest połączona z trzema pamięciami podręcznymi, pamięcią danych, wstępnego pobierania i zapisu.

Pamięć podręczna danych (ang. *data cache*) to czterokanałowa, asocjacyjna pamięć podręczna pierwszego poziomu o rozmiarze wiersza 32 bajty i pojemności 64KB. Pamięć podręczna pobierania wstępnego (ang. *prefetch cache*) o pojemności 2KB to pamięć wykorzystywana na potrzeby specjalnych rozkazów pobierania wstępnego. W tym miejscu niezbędne jest jednak wsparcie ze strony kompilatora. Na etapie kompilacji, jeżeli kompilator przewiduje, że dane słowo pamięci będzie wykorzystywane wielokrotnie, może wstawić do programu rozkaz pobierania wstępnego. W czasie wykonywania programu powoduje to załadowanie odpowiedniego słowa do pamięci *prefetch cache*, przyspieszając w ten sposób dostęp do tego słowa danych kilka instrukcji dalej.

Pamięć podręczna zapisu (ang. *write cache*) ma pojemność 2KB. Jej jedynym zadaniem jest efektywne wykorzystanie magistrali pamięci podręcznej L2 do zapisu danych.

Układ zawiera również logikę sterowania dostępem do pamięci. Jest ona podzielona na 3 części, interfejs systemowy, sterownik pamięci podręcznej drugiego poziomu i sterownik pamięci głównej. Interfejs systemowy łączy procesor z pamięcią za pośrednictwem 128-bitowej magistrali. Interfejs ten realizuje wszystkie żądania kierowane na zewnątrz, oprócz tych kierowych do pamięci L2. Interfejs ten został tak zaprojektowany, aby z jednej pamięci mogło korzystać wiele procesorów.

Sterownik pamięci podręcznej drugiego poziomu jest połączony z jednolitą pamięcią L2. Różnicą w stosunku do procesora Intel Pentium 4 jest to, że pamięć drugiego poziomu w procesorach Suna znajduje się poza procesorem jako oddzielny układ. Pozwala to na stosowanie większej ilości pamięci, jest jednak wolniejsze z powodu większej odległości pomiędzy pamięcią a procesorem.
Sterownik pamięci głównej odwzorowuje 64-bitowe adresy wirtualne na 43-bitowe adresy fizyczne.

Potok procesora UltraSPARC III Cu ma 14 stopni. Zostały one przedstawione na Rys. 6.7. Kolejne stopnie zostały nazwane literami alfabetu.

Stopień *A* (ang. *address generation*) odpowiada za generowanie adresu. Określa adres kolejnego rozkazu do pobrania. Najczęściej jest to adres kolejnego rozkazu, jednak w przypadku instrukcji warunkowych, przerwań, wyjątków oraz skoku sekwencja kolejnego wykonywania rozkazów może być zaburzona. Przewidywanie rozgałęzień wymaga więcej niż jednego cyklu zegara, rozkaz umieszczony w pamięci bezpośrednio po skoku warunkowym jest zawsze wykonywany, bez względu na to, czy skok nastąpi czy nie. Często w tym miejscu wstawia się rozkaz `nop` jako wypełnienie powstałej szczeliny opóźnienia (ang. *delay slot*). Szczelinę opróżnienia można również wykorzystać przy wywołaniach procedur.

Stopień *P* (ang. *preliminary fetch*) wykorzystuje adres dostarczony przez stopień *A* aby rozpocząć pobieranie rozkazów z pamięci podręcznej L1. W przypadku braku chybień, stopień *P* jest w stanie pobrać do 4 rozkazów. W tym stopniu potoku sprawdzana jest także wartość tablicy rozgałęzień w celu ustalenia, czy wśród pobieranych rozkazów są skoki warunkowe. Jeżeli tak, dokonywane jest prognozowanie skoku.

Stopień *F* (ang. *fetch*) kończy pobieranie rozkazów do pamięci.

Stopień *B* (ang. *branch target*) dekoduje pobrane rozkazy. Jeżeli występują wśród nich skoki prognozowane jako wykonywane, informacja ta jest na tym stopniu dostępna i jest przekazywana do stopnia *A*, który kieruje dalszym pobieraniem rozkazów.

Stopień *I* (ang. *instruction group formation*) łączy otrzymywane rozkazy w grupy zależnie od tego, które z sześciu jednostek funkcjonalnych rozkazy te wykorzystują: dwie stałopozycyjne jednostki ALU, dwie zmiennopozycyjno/graficzne jednostki ALU, potok rozgałęzień oraz układ odpowiedzialny za operacje ładowania i zapisywania danych z/do pamięci.

Stopień *J* (ang. *instruction stage grouping*) usuwa rozkazy z kolejki i przygotowuje je do przydzielenia jednostkom wykonawczym w kolejnym cyklu. O przypisaniu rozkazów do jednostek wykonawczych decyduje zarówno charakter rozkazu jak i dostępność jednostek wykonawczych.

Stopień *R* (ang. *register access*) wyszukuje rejestry wymagane przez rozkazy stałopozycyjne i przekazuje żądania dotyczące rejestrów zmiennopozycyjnych do zestawu rejestrów zmiennopozycyjnych. Jeżeli potrzebny rejestr nie jest dostępny, rozkaz jest wstrzymywany i oczekuje aż poprzedni rozkaz wykona się, a rejestr się zwolni.

Stopień *E* (ang. *execution*) jest jednostką wykonawczą procesora. Większość rozkazów arytmetyczno logicznych może być ukończona w jednym cyklu zegara. Operacje ładowania i zapamiętywania są inicjowane, ale nie zostają ukończone. W tym stopniu potoku przetwarzane są też rozkazy skoków warunkowych i określane jest, czy skok będzie wykonany. Jeżeli prognoza się nie powiedzie, potok jest opróżniany.

Stopień *C* (ang. *cache*) to stopień, na którym kończy się operacje dostępu do pamięci podręcznej pierwszego poziomu.

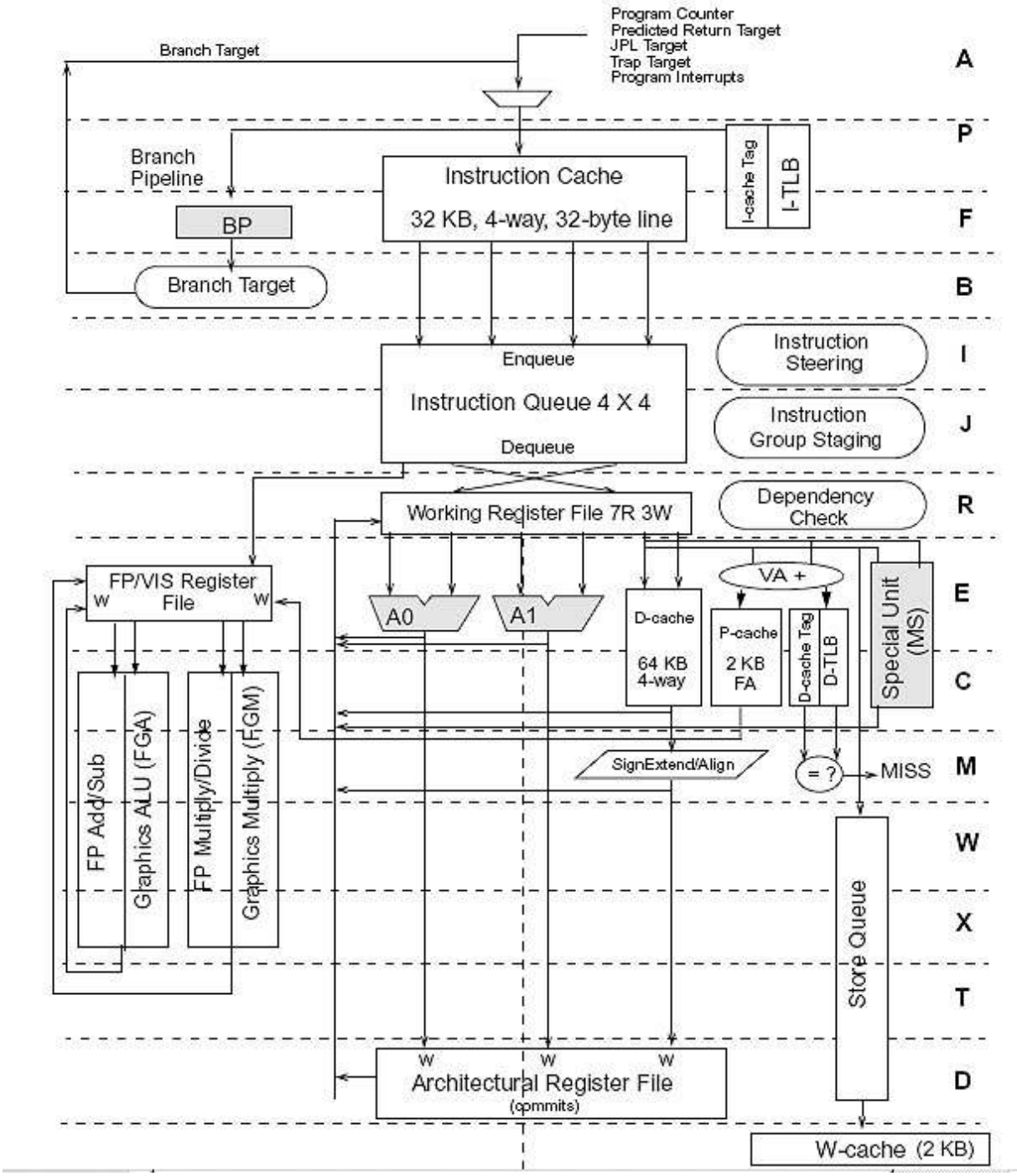
Stopień *M* (ang. *miss*) rozpoczyna przetwarzanie słów danych, które nie zostały odnalezione w pamięci L1. Jeżeli słowo nie jest dostępne w pamięci L1, przeszukuje się pamięć L2, a w przypadku ponownego chybienia procesor odwołuje się do pamięci operacyjnej.

Stopień *W* (ang. *write*) zapisuje wyniki operacji do zestawu rejestrów roboczych.

Stopień *X* (ang. *extended*) kończy wykonywanie większości operacji zmiennopozycyjnych i graficznych.

Stopień *T* (ang. *trap*) wykrywa pułapki stało- i zmiennopozycyjne. Po wystąpieniu pułapki, stopień ten odpowiada za zakończenie wszystkich wcześniej wykonywanych rozkazów i nie rozpoczynanie kolejnych.

Stopień *D* (ang. *done*) rejestry stało- i zmiennopozycyjne są zapisywane do właściwych zestawów rejestrów architektury.



Rys. 6.7 Schemat budowy procesora UltraSPARC III Cu z zaznaczonym potokiem [21]

W Tabela 24 zebrano podstawowe informacje o procesorach 8051, Intel Pentium 4 oraz UltraSPARC III Cu. Różnice w poszczególnych cechach procesorów, wynikają przede wszystkim z różnych zastosowań tych procesorów i całkowicie odmiennych stawianych im wymagań.

Tabela 24 Zestawienie właściwości procesorów Intel 8051, Intel Pentium 4, UltraSPARC III

	Intel 8051	Intel Pentium 4	UltraSPARC III Cu
Zastosowanie	systemy wbudowane (ang. *embedded* systems)	komputery osobiste (ang. *personal computer*)	wysokowydajne serwery wieloprocesorowe
Rok zaprojektowania	1983	2000	2000
Ilość tranzystorów	60 000	42 – 55 milionów	29 milionów
Taktowanie[5]	*nie znaleziono*	1.5 GHz – 3.2 GHz	0.6 GHz – 1.2 GHz
Szerokość ścieżek	*nie znaleziono*	0.18µ – 0.09µ	0.18µ – 0.13µ
Lista rozkazów	CISC (ang. *complex instruction set computer*)	CISC	RISC (ang. *rationalized / reduced instruction set computer*)
Pamięć wirtualna	brak	tak	tak
Pamięć podręczna	brak	tak	tak
Potokowanie (*pipeling*)	brak	tak jednostka czołowa, sterowanie niekolejnym wykonywaniem rozkazów, jednostka wykonawcza	tak 14 stopniowy *A, P, F, B, I, J, R, E, C, M, W, X, T, D*
Architektura ISA	Proste rozkazy, które można wykonać w jednym cyklu procesora, jednak procesor zaliczany jest do rodziny CISC	Translacja rozkazów na mikrooperacje przez układ czołowy procesora	Instrukcje są mikrooperacjami
Branch Prediction	brak	tak	tak
Kolejność wykonywania rozkazów	rozkazy wykonywane są sekwencyjnie	procesor może zmienić kolejność wykonywania rozkazów	rozkazy wykonywane są sekwencyjnie
Zastosowane technologie	powielone zestawy rejestrów w celu szybkiego przełączenia kontekstu	NetBurst	Ultra Port Architecture

[5] Porównanie prędkości pracy procesorów należących do rodzin RISC i CISC tylko przy pomocy prędkości taktowania zegara jest zadaniem bardzo trudnym, a sama metodologia pomiaru niewłaściwa

7 Wprowadzenie do języka asembler

Język asembler jest bardzo ściśle powiązany z poziomem maszynowym ISA przedstawionym w poprzednim rozdziale. Stanowi jego abstrakcyjną, programową implementację dając programiście łatwy sposób sterowania komputerem. Języki asemblera powstały w celu ułatwienie kodowania programów. Używanie etykiet dla adresów oraz kodów mnemonicznych dla kodów binarnych instrukcji jest o wiele prostsze niż używanie długich sekwencji liczb binarnych.
W języku asemblera każda instrukcja źródłowa odpowiada dokładnie jednemu rozkazowi języka maszyny. Translacja języka asembler na język maszynowy jest odwzorowaniem typu jeden do jednego.
Instrukcje asemblera dla różnych procesorów, pomimo różnic wynikających z odmiennej budowy procesorów mają wiele cech wspólnych. W większości przypadków instrukcja asemblera składa się z czterech pól: etykiety (ang. *label*), kodu instrukcji (ang. *operation code*), operandów (ang. *operands*) i komentarza (ang. *comment*). Każde z tych pól jest opcjonalne.
Zadaniem etykiety jest symboliczne wyróżnienia adresów pamięci zawierających kod wykonywalny. Dzięki temu można wykonywać skoki do tych adresów nie w sposób bezwzględny zależny od ułożenia poszczególnych linii kodu asemblera, lecz w sposób bardziej pośredni odnosząc się do tekstowych identyfikatorów.
Pole kodu operacji zawiera mnemoniczny kod rozkazu maszynowego, albo polecenie samego asemblera. Mnemonik, czyli sugestywny skrót literowy pochodzi głównie od pierwszych liter wyrazów w języku angielskim opisującym czynność wykonywaną przez instrukcję, np.: `add` (ang. *to add*). Zestaw i znaczenie używanych nazw jest specyficzny dla konkretnego asemblera.
Wśród instrukcji istnieją bezargumentowe (`nop` – instrukcja pusta), jednoargumentowe (`incl %eax` – zwiększa wartość rejestru o jeden), dwuargumentowe (`movl %eax, %ebx` – kopiuje wartość z rejestru `eax` do rejestru `ebx`) oraz popularne w asemblerze SPARC instrukcje trzyargumentowe, w których przeznaczenie dla wyniku podaje się w sposób jawny (`add %r1, %r2, %r3` – dodaj wartość rejestru `r1` do wartości rejestru `r2` i wynik zapisz w rejestrze `r3`). Dla instrukcji mających argumenty, niezbędne jest podanie ich operandów. Operand można podać w jeden z wielu dostępnych sposobów adresowania. Można również wyspecyfikować typ operandu jako np.: bajt, znak, słowo itd… Typ operandów w asemblerze dla procesorów Intel w notacji AT&T podaje się zazwyczaj jako sufix do mnemonika, np. `movb %ax, %bx` kopiuje bajt między młodszą częścią rejestru `eax` zwaną `ax` a młodszą częścią rejestru `ebx` zwaną `bx`, podczas gdy instrukcja `movl %eax, %ebx` kopiuje całe słowo pomiędzy rejestrami.
Instrukcja asemblera może kończyć się opisem działania w języku naturalnych, zupełnie ignorowanym przez komputer – komentarzem. Aby komentarz mógł być rozróżniony od właściwego kodu źródłowego, umieszcza się go po specjalnym znaku, # dla asemblera dla procesorów Intel w notacji AT&T oraz po znaku ! w asemblerze SPARC. Komentarz jest usuwany przy translacji do kodu maszynowego, dlatego nie wpływa ani na rozmiar kodu wynikowego ani na jego prędkość działania. Natomiast dobrze napisany komentarz może bardzo ułatwić zrozumienie działania programu oraz intencji programisty.
Najprostszy asembler dwukrotnie wczytuje dane wejściowe. W trakcie pierwszego przebiegu wszystkie identyfikatory oznaczające adresy są zapisywane w tablicy symboli. Każdemu identyfikatorowi, w chwili jego napotkania, jest przypisywany adres pamięci. W czasie drugiego przebiegu asembler wczytuje dane wejściowe ponownie. Tym razem tłumaczy kod każdej operacji na sekwencję bitów reprezentującą ją w kodzie maszynowym oraz tłumaczy każdy identyfikator na przypisany mu adres w pierwszym przebiegu. Wynikiem drugiego przebiegu jest zwykle tzw. kod maszynowy przemieszczalny, czyli taki, który może być załadowany pod dowolne miejsce w pamięci.

Program w kodzie maszynowym jest bardzo ściśle zależy od procesora, na którym zostanie uruchomiony. Determinuje to zbiór użytych rozkazów, wykorzystanie rejestrów oraz sposób dostępu do pamięci.
Fazie translacji kodu asemblera do kodu maszynowego powierza się wiele zadań. Kod wyjściowy musi być poprawny i wysokiej jakości, co oznacza, że powinien efektywnie wykorzystywać zasoby komputera np.: rejestry procesora. Pożądanym zachowaniem podczas translacji jest również optymalizacja kodu. Z matematycznego punktu widzenia, problem generowania optymalnego kodu jest nierozstrzygalny, dlatego używa się w tym celu metod heurystycznych.

7.1 Format rozkazów

Format rozkazów procesora 8051

Procesor 8051 ma sześć prostych formatów rozkazów.

1. Format zawiera tylko kod operacji. Jest on wykorzystywany np.: w instrukcjach bezargumentowych, takich jak inkrementowanie zawartości rejestru – *inc*.

8b kod operacji

2. Kod operacji ma 5 bitów, z pozostałe 3 bity to numer rejestru. Z tego formatu korzystają instrukcje, które wykonują operacje na zawartości akumulatora i jednego rejestru, np. dodawanie *add*, kopiowanie *mov* itp.

5b kod operacji	3b rejestr

3. Operand ma 1 bajt, może nim być stała, offset lub numer bitu.

8b kod operacji	8b operand

4. Rozkaz zawiera 11 bitowy adres, który umożliwia adresowanie 2048 bajtów. Instrukcje używa się, gdy nie jest podłączona pamięć zewnętrzna.

5b kod operacji	11b adres

5. Rozkaz zawiera 16 bitowy adres, umożliwiający zaadresowanie do 64KB bajtów, format instrukcji używany jest do adresowania pamięci zewnętrznej.

8b kod operacji	16b adres

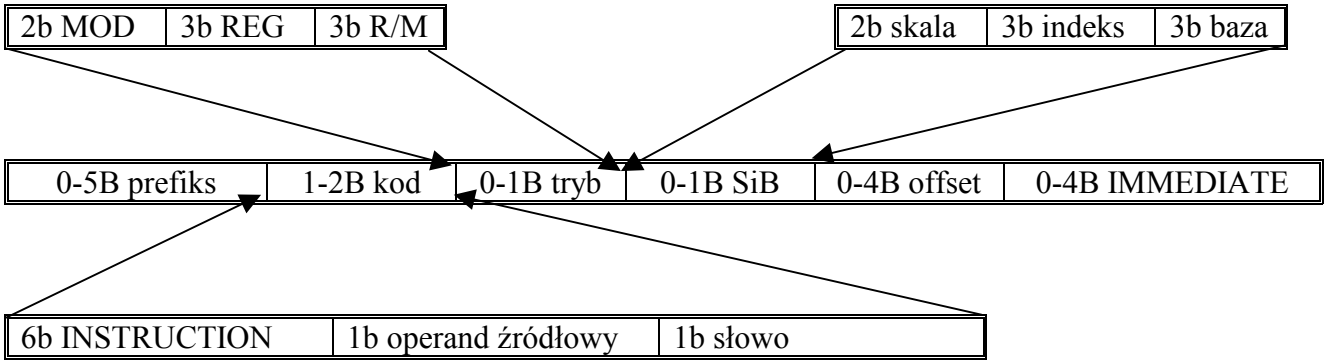
6. Rozkaz zawiera dwa ośmiobitowe operandy. Korzystają z niego instrukcje takie jak, kopiowanie 8-bitowej stałej do pamięci.

8b kod operacji	8b operand	8b operand

Format rozkazów procesora Intel Pentium 4

Procesor Intel Pentium 4 ma wiele sposobów adresowania, które znacznie się różnią. Używanych jest do sześciu pól o zmiennej długości, z których pięć jest opcjonalnych. Wynika to z długiej ewolucji architektury oraz próby utrzymania kompatybilności wstecznej.
Schemat rozkazu zaprezentowano na Rys. 7.1.

Bajt prefiksu (ang. *prefix byte*) to dodatkowy kod operacji umieszczany przed właściwym rozkazem w celu zmiany jego działania.
MOD – określa jeden z czterech trybów adresowania
REG – określa numer rejestru, w którym znajduje się operand
R/M – oznacza sposób wyznaczenie miejsca operandu
SiB (ang. *Scale, Index, Base*) pole dodane w celu kompatybilności wstecznej, umożliwia różnież dodanie nowych rozkazów



Rys. 7.1 Format rozkazu procesora Intel Pentium 4

Format rozkazów procesora UltraSPARC III Cu

Na poziomie maszynowym procesora UltraSPARC III Cu stosowane są wyłącznie rozkazy 32-bitowe, wyrównane w pamięci.
Pierwsze dwa bity umożliwiają określenie formatu rozkazu. Warto zaznaczyć, że dwa bity umożliwiają jednoznaczne zidentyfikowanie czterech formatów. W rzeczywistości format 1 i 2 mają takie same pierwsze dwa pola bitu a ich rozpoznanie następuje po bicie 13, który w przypadku formatu *trójadresowego* ma wartość 0 natomiast w formacie *stałym* wartość równą 1.
Sześciobitowe pole kod operacji umożliwia zakodowanie 2^6, czyli 64 różnych operacji.

1. Format trójadresowy

2b	5b przeznaczenie	6b kod operacji	5b źródło	1b 0	8b FP-OP	5b SRC2

2. Format stały

2b	5b przeznaczenie	6b kod operacji	5b źródło	1b 1	13b stała natychmiastowa

3. Format SETHI stosowany jest do adresowania 32-bitowych pól. Nie jest możliwe zaadresowanie 32 bitowej stałej w jednym 32 bitowym rozkazie, dlatego aby uzyskać ten efekt należy wykonać dwie instrukcje. Pierwsza z nich z formacie SETHI kopiuje 22 najbardziej znaczące bity do rejestru, kolejne 10 bitów mniej znaczących bitów ustawia się za pomocą drugiego rozkazu.

2b	5b przeznaczenie	3b OP	22b stała natychmiastowa

4. Format BRANCH jest wykorzystywany do nieprognozowanych skoków warunkowych skojarzonych z instrukcjami warunkowymi. Bit *A* służy do uniknięcia szczeliny opóźnienia.
   Skoki prognozowane korzystają z tego samego formatu rozkazów, w którym jednak 22 bitowe przesunięcia zredukowano do pola 19 bitowego. Jeden z zaoszczędzonych w ten sposób bitów został wykorzystany do prognozowania skoku a dwa służą do określania wykorzystywanego zestawu kodów warunkowych

2b	1b A	4b warunek	3b OP	22b przesunięcie względem licznika rozkazów

5. Format CALL jest wykorzystywany przy instrukcji *call* służącej do wywoływania procedur. Pole adresu zawiera adres docelowy podzielony przez 4. Umożliwia to wywołania procedur oddalonych od bieżącego rozkazu o co najwyżej $\pm 2^{31}$ bajtów.

2b	30b przesunięcie względem licznika rozkazów

7.2 Typy danych

Typy danych można podzielić na dwie kategorie, numeryczne i nienumeryczne. Podstawowy typ danych numerycznych to liczba całkowita. Liczby całkowite mogą mieć różną długość od 8 do 64 bitów, mogą być ze znakiem lub bez niego. Częstym sposobem reprezentacji znaku jest przypisanie ustalonej wartości do jednego z bitów liczby kosztem zmniejszenia zakresu liczby.
Bardziej złożonym formatem liczb jest format zmiennopozycyjny, wykorzystywany do zakodowania liczb rzeczywistych. Typowe rozmiary liczby zmiennopozycyjnych są zazwyczaj dużo większe niż dla liczb całkowitych i mieszczą się w granicach od 32 do 128 bitów.

Typy danych dostępne dla procesorów 8051, Intel Pentium 4 oraz UltraSPARC III Cu przedstawiono w Tabela 25.

Tabela 25 Dostępne tryby adresowania dla procesorów 8051, Intel Pentium 4 oraz UltraSPARC III Cu

	Bitowy	Liczba całkowita ze znakiem	Liczba całkowita bez znaku	Zmiennopozycyjny
8051	tak	8 bitowy	brak	brak
Intel Pentium 4	nie	8 bitowy 16 bitowy 32 bitowy	8 bitowy 16 bitowy 32 bitowy	32 bitowy 64 bitowy
UltraSPARC III Cu	nie	8 bitowy 16 bitowy 32 bitowy	8 bitowy 16 bitowy 32 bitowy	32 bitowy 64 bitowy 128 bitówy

7.3 Adresowanie

Adresowanie natychmiastowe (ang. *immediate addresing*)

Adresowanie natychmiastowe jest najprostszą metodą adresowania, w którym operand jest jawnie zapisany. W rzeczywistości nie podaje się adresu operandu tylko jawnie specyfikuje jego wartość. Wadą tego rozwiązania jest to, że w ten sposób można używać tylko stałe.
Na Rys. 7.2 pokazano kod asemblera z adresowaniem natychmiastowym dla architektur Intela oraz Suna.

UltraSPARC III Cu	`or 4, %r1, %r1`
Intel x86 (notacja MASM)	`mov eax, 4`
Intel X86 (notacja AT&T)	`movl $4, %eax`

Rys. 7.2 Adresowanie natychmiastowe w asemblerze dla procesorów Intel oraz Sun

Adresowanie bezpośrednie (ang. *direct addressing*)

Innym sposobem adresowania jest adresowanie bezpośrednie. Polega na jawnym zapisaniu adresu operandu. Adresowanie te jest wygodnym sposobem dostępu do zmiennych globalnych, których adres jest niezmienny w trakcie działania programu. Adres można opcjonalnie poprzedzić segmentem, do którego się odwołujemy.

UltraSPARC III Cu	`niedostępne`
Intel x86 (notacja MASM)	`mov eax, [0xFFFFFF]`
Intel X86 (notacja AT&T)	`movl 0xFFFFFF, %eax`

Rys. 7.3 Adresowanie bezpośrednie w asemblerze dla procesorów Intel oraz Sun

Adresowanie rejestrowe (ang. *register addressing*)

W adresowaniu rejestrowym, wskazuje się rejestr, który przechowuje operand. Ma to taka samą wadę jak adresowanie bezpośrednie, adres operandu trzeba jawnie zakodować na stałe w programie.

UltraSPARC III Cu	`or %g0, %r1, %r1`
Intel x86 (notacja MASM)	`mov ebx, eax`
Intel X86 (notacja AT&T)	`movl %eax, %ebx`

Rys. 7.4 Adresowanie rejestrowe w asemblerze dla procesorów Intel oraz Sun

Adresowanie pośrednie rejestrowe (ang. *indirect addressing mode*)

Ten rodzaj adresowania jest pierwszym, w którym nie koduje się adresu operandu na stałe w kodzie programu. Adres jest zapisany w jednym z rejestrów, w kodzie natomiast wskazuje się w którym. Zasadniczą różnicą w stosunku do adresowania rejestrowego jest to, że rejestr zawiera adres operandu a nie wartość operandu. Rejestr, który zawiera adres operandu nazywa się wskaźnikiem (ang. *pointer*). Ta sama instrukcja programu może adresować różne miejsca w pamięci, wystarczy zmienić cel, na który wskazuje wskaźnik.

UltraSPARC III Cu	`ld [%r1], %r2`
Intel x86 (notacja MASM)	`mov ebx, [eax]`
Intel X86 (notacja AT&T)	`movl (%eax), %ebx`

Rys. 7.5 Adresowanie pośrednie rejestrowe w asemblerze dla procesorów Intel oraz Sun

Adresowanie indeksowe (ang. indexed addressing)

Adresowanie indeksowe jest bardzo efektywne przy odwoływaniu się do słów pamięci o znanym przesunięciu względem adresu przechowywanego w rejestrze. Rejestr zawiera adres początku bloku danych, do którego dodaje się przesunięcie (ang. *offset*) uzyskując dostęp do wybranych słów pamięci.

Intel x86 (notacja MASM)	`mov eax, [ 0xFFFFFF + ecx * 1 ]`
Intel X86 (notacja AT&T)	`movl 0xFFFFFF( , %ecx, 1 ), %eax`

Rys. 7.6 Adresowanie indeksowe w asemblerze dla procesorów Intel oraz Sun

Adresowanie indeksowe z wartością bazową (ang. *based indexed addressing*)

Niektóre procesory dysponują trybem adresowania, w którym adres pamięci obliczany jest przez zsumowanie wartości dwóch rejestrów i opcjonalnego przesunięcia. Jeden rejestr oznacza adres podstawowy, drugi indeks. Indeks jest mnożony przez rozmiar słowa a do całości można dodać opcjonalnie przesunięcie. Schemat tego adresowania to

```
liczba( %baza, %index, skala]
```

w notacji AT&T oraz

```
[baza + index * skala + liczba]
```

w notacji MASM. Baza i indeks muszą być przechowywane w rejestrach. Wszystkie parametry adresowania są opcjonalne. Warto zauważyć, że adresowanie indeksowe jest szczególnym przypadkiem adresowania indeksowego z wartością bazową, w którym pomija się rejestr bazowy.

Intel x86 (notacja MASM)	`movl eax, [ 0xFFFFFF + ebx + ecx * 1 ]`
Intel X86 (notacja AT&T)	`movl 0xFFFFFF( %ebx, %ecx, 1 ), %eax`

Rys. 7.7 Adresowanie indeksowe z wartością bazową w asemblerze dla procesorów Intel oraz Sun

7.4 Instrukcje procesora Intel Pentium 4

Instrukcje asemblera dla procesorów Intel x86 w notacji AT&T składają się z mnemonika rozkazu i argumentów rozkazu. Większość instrukcji może działać na różnych typach danych, takich jak bajt słowo. Typ danych, na których pracuje rozkaz określa się sufixem rozkazu, b – bajt, l – słowo.

Tabela 26 Instrukcje transferu danych asemblera w notacji AT&T

OPERACJE TRANSFERU DANYCH	
INSTRUKCJA	OPIS I PRZYKŁAD
`mov[bl]` *`src, dest`*	Kopiuje wartość ze źródła *`src`* do celu *`dest`*. Instrukcja `movl %eax, %ebx` kopiuje zawartość rejestru `eax` do rejestru `ebx` Instrukcja `movl (%eax), %ebx` używa adresowania pośredniego, wartość spod adresu zapisanego w rejestrze `eax` jest kopiowana do rejestru `ebx` Instrukcja `movl $123, %eax` używa adresowania natychmiastowego, wartość `123` jest kopiowana do rejestru `eax` Instrukcja `movl 123, %eax` używa adresowania bezpośredniego, wartość spod adresu `123` jest kopiowana do rejestry `eax` Instrukcja `movl 5(%ebp, %ecx, 1), %eax` używa adresowania indeksowego z wartością bazową, wartość spod adresu `5+%ebp+1*%ecx` ładowana jest do rejestru `eax`. Rejestr `ebp` wskazuje wartość bazową, rejestr `ecx` może być użyty w roli indeksu pętli.
`lea[bl]` *`src, dest`*	Kopiuje adres efektywny ze źródła *`src`* do celu *`dest`*. Instrukcja `leal 5(%ebp, %ecx, 1), %eax` ładuje adres `5+%ebp+1*%ecx` do rejestru `%eax`.

`push[bl]` *`src`*	Umieszcza wartość źródła *`src`* na stosie
	Instrukcja `pushl %eax`
	umieszcza zawartość rejestru `eax` na stosie
	Instrukcja jest równoważna sekwencji
	`subl $4, %esp`
	`movl %eax, (%esp)`

`pop[bl]` *`dest`*	Przenosi wartość ze stosu do celu *`dest`*
	Instrukcja `popl %eax`
	przenosi wartość ze stosu do rejestru `eax`
	Instrukcja jest równoważna sekwencji
	`movl (%esp), %eax`
	`add $4, %esp`

`xchg[bl]` `dest1, dest2`	Wymienia zawartości `dest1` i `dest2`.

Procesor Intel x86 został wyposażony w wiele instrukcji pozwalających dokonywać obliczeń na liczbach całkowitych, instrukcji pozwalających stosować funkcję logiczne na bitach jak również operować poszczególnymi bitami. Tabela 27zawiera listę najczęściej stosowanych instrukcji, wraz z ich opisem i przykładem użycia.

Tabela 27 Instrukcje arytmetyczno - logiczne asemblera w notacji AT&T

OPERACJE ARYTMETYCZNO - LOGICZNE	
INSTRUKCJA	OPIS I PRZYKŁAD
`addl` *src, dest*	Dodaje wartość ze celu *dest* do wartości źródła *src*, wynik zapisany będzie jako wartość celu *dest* Instrukcja `addl $4, %eax` dodaje do wartości rejestru `eax` liczbę cztery
`Subl` *src, dest*	Odejmuje od wartości celu *dest* wartość źródła *src*, wynik zapisany będzie jako wartość celu *dest* Instrukcja `sub $8, %eax` odejmie od wartości rejestru `eax`, wynik zapisze w `eax`.
`[i]mull` *src, dest*	Mnoży wartość (bez znaku `imull`, ze znakiem `mull`) celu *dest* przez wartość źródła *src*, drugi argument jest opcjonalny – domyślną wartością jest rejestr `eax`, wynik zapisywany jest w podwójnym słowie `%edx:%eax` Instrukcja `imull $2` mnoży wartość rejestru `eax` przez dwa.
`[i]divl` *src, dest*	Dzieli wartość (bez znaku `idivl`, ze znakiem `divl`) podwójnego słowa `%edx:%eax` przez wartość rejestru wskazanego jako drugi operand *dest*.
`negl` *dest*	Neguje wartość celu *dest* w kodzie dopełnienie do dwóch. Jeśli rejestr ma maskę bitową `1010101010101010`, instrukcja `negl %eax` zaneguje najstarszy bit, rejestr `eax` będzie mieć maskę bitową `0010101010101010`.

`incl` *dest*	Inkrementuje wartość celu *dest*
	Sekwencja instrukcji
	`movl $0,%eax`
	`incl %eax`
	ustawia wartość rejestru `eax` na 1.

`decl` *dest*	Dekrementuje wartość celu *dest*
	Sekwencja instrukcji
	`movl $1, %eax`
	`decl %eax`
	ustawia wartość rejestru `eax` na 0.

`andl` *src, dest*	Koniunkcja bitowa źródła *src* z celem *dest,* wynik zapisuje w *dest* Instrukcja `andl %eax,%ebx` oblicza koniunkcję bitową wartości rejestrów `eax,ebx`.
`orl` *src, dest*	Alternatywa bitowa źródła *src* z celem *dest*, wynik zapisuje w *dest* Instrukcja `orl %eax,%ebx` oblicza alternatywę bitową wartości rejestrów `eax, ebx`.
`notl` *dest*	Neguje wszystkich bitów celu. Jeśli rejestr ma maskę bitową `1111111100000000`, instrukcja `notl %eax` neguje wszystkie bity a rejestr `eax` będzie mieć maskę bitową `0000000011111111`.
`xorl`	

Tabela 27 Instrukcje arytmetyczno - logiczne asemblera w notacji AT&T

OPERACJE ARYTMETYCZNO - LOGICZNE	
INSTRUKCJA	OPIS I PRZYKŁAD
src, dest	Bitowa różnica symetryczna źródła z celem, wynik zapisze w *dest* Instrukcja `xorl %eax,%ebx` oblicza bitową różnicę symetryczną wartości rejestrów `eax,ebx`.

`roll`
number, dest
 	
`rorl`
number, dest
	Instrukcja wykonuje rotację bitów celu *dest* w lewo (`roll`) lub prawo (`rorl`). Liczba bitów, o jaką wykonana będzie rotacja *number* jest zapisana albo przy pomocy adresowania natychmiastowego, lub w młodszej części rejestru `ecx`, czyli `cx`.
	Jeśli rejestr `eax` ma maskę bitową `1111000011110000`,
	instrukcja `roll $1, %eax`
	wykona rotację bitów w lewo o jedną pozycję, najstarsze bity staną się najmłodszymi, a rejestr `eax` będzie mieć maskę bitową `1111000111100001`.

`aall`
number, dest
 	
`aarl`
number, dest
	Arytmetyczne przesunięcie wartości celu *dest* w lewo (`sall`) lub prawo (`sarl`) o zadaną liczbę bitów `number`.
	Jeśli rejestr `eax` zawiera maskę bitową `1010101010101010`
	instrukcja `sall $1, %eax`
	zachowa najstarszy bit na swoim miejscu
	pozostałe bity przesunie o jedną pozycję w lewo
	a nowo powstały najmłodszy bit wypełni wartością zera,
	rejestr `eax` będzie mieć maskę bitową `1101010101010100`.
	Arytmetyczne przesunięcie w lewo lub w prawo, nie zmienia wartości najstarszego bitu, czyli nie zmienia znaku wartości.

`shll`
number, dest
 	
`shrl`
number, dest
	Przesunięcie wartości celu *dest* w lewo (`shll`) lub prawo (`shrl`) o zadaną liczbę bitów `number`
	Jeśli rejestr `eax` zawiera maskę bitową `1010101010101010`
	instrukcja `sall $1, %eax`
	przesunie wszystkie bity o jedną pozycję w lewo
	a nowo powstały najmłodszy bit wypełni wartością zera,
	rejestr `eax` będzie mieć maskę bitową `0101010101010100`.
	Arytmetyczne przesunięcie w lewo lub w prawo, zmienia wartość najstarszego bitu, czyli zmienia znak wartości.

Tabela 28 Instrukcje skoków asemblera w notacji AT&T

INSTRUKCJE SKOKÓW	
INSTRUKCJA	OPIS I PRZYKŁAD
`cmpl` *value1,* *value2*	Instrukcja porównuje dwie wartości. Wynik porównania zapisywany jest w rejestrze flag skąd odczytuje się go instrukcjami `je,jg,jge...`

`call` *label*	Instrukcja wywołuje procedurę. Adres funkcji, oprócz etykiety tekstowej, może być również opisany przez wartość rejestru `eax`. Należy wtedy jednak poprzedzić rejestr `eax` znakiem *.
	Jeżeli wartości etykiety `procedura` i rejestru `eax` są sobie równe, instrukcje
	`call procedura`
	`call *eax`
	są sobie równoważne, i wywołują odpowiednią procedurę.

`int` *number*	Wywołuje przerwanie o podanym numerze. Przerwanie pod adresem `0x80` w systemie LINUX wykonuje funkcję systemową. Numer funkcji podawany jest w rejestrze `eax`, jej opcjonalny parametr w rejestrze `ebx,ecx`.Jeżeli funkcja systemowa `exit` ma wartość `1`,

Tabela 28 Instrukcje skoków asemblera w notacji AT&T

INSTRUKCJE SKOKÓW	
INSTRUKCJA	OPIS I PRZYKŁAD

	sekwencja instrukcji zakończy działanie programu z kodem `0`.
	`movl $1, %eax`
	`movl $0, %ebx`
	`int $0x80`

`jxx` *`label`*	Zbiór instrukcji skoków warunkowych. Po literze `j` wystepuje sufix określający warunek logiczny, który w przypadku jego spełniania warunkuje skok do etykiety *`label`*. Warunek może mieć następującą postać.
	`mp` – skok bezwarunkowy do etykiety *`label`*.
	Instrukcja `jmp loop_end`
	zawsze przeskoczy do etykiety `loop_end`.
	`e` – skok do etykiety *`label`*, jeżeli porównywane wartości są równe.
	Sekwencja instrukcji
	`movl $2, %eax`
	`cmlp %eax, $2`
	`je koniec_programu`
	przejdzie do etykiety `koniec_programu`, ponieważ wartość rejestru `eax` i stała `2` są sobie równe.
	`g[e]` – skok do etykiety *`label`*, jeżeli druga z porównywanych wartości jest większa (`jg`) lub większa – równa (`jge`) pierwszej.
	Sekwencja instrukcji
	`movl $2, %eax`
	`cmpl %eax, $3`
	`je koniec_programu`
	przejdzie do etykiety `koniec_programu`, ponieważ druga z porównywanych wartości jest większa lub równa niż pierwsza.
	`l[e]` – skok do etykiety *`label`*, jeżeli druga z porównywalnych wartości jest mniejsza (`je`) lub mniejsza – równa (`jle`) od pierwszej.

7.5 Instrukcje procesora UltraSPARC III Cu

Tabela 29 Instrukcje transferu danych asembler

ŁADOWANIE Z PAMIĘCI	
INSTRUKCJA	OPIS
`ld[us][bhw] [src], dest`	Instrukcja ładuje wartość z pamięci do rejestru. Adres może być pojedynczym rejestrem, sumą dwóch rejestrów lub sumą rejestru i stałej. Opcjonalny przyrostek `u` oznacza liczbę bez znaku, `s` liczbę ze znakiem, `b` bajt, `h` pół słowa, `w` całe słowo. Instrukcja `ldub [%r1], %r2` ładuje liczbę bez znaku o szerokości bajta z rejestru `r1` do rejestru `r2`.
`st[us][bhw] [src], dest`	Instrukcja zapisuje wartość z rejestru do pamięci. Źródło może być pojedynczym rejestrem, sumą dwóch rejestrów lub sumą rejestru i stałej. Instrukcja `stsw %r2, [%r1]` ładuje liczbę ze znakiem o szerokości całego słowa z rejestru `r2` do rejestru `r1`.
`swap[us][bhw] [src], dest`	Instrukcja zamienia wartość pamięci z wartością rejestru. Instrukcja `swap [%r1], r2` zamienia wartości rejestrów `r1` i `r2`.

`sethi src, dest`	Instrukcja `sethi` ustawia bity od 10 do 31 w wartości celu. Rozkaz ten jest niezbędny, ponieważ nie ma możliwości wpisania do rejestru 32-bitowej liczby przy użyciu 32 bitowej instrukcji. Pojemność instrukcji jest za mała, żeby przechować 32 bitową liczbę oraz kod instrukcji, adres rejestru itd.… Proces ładowania liczby 32 bitowej rozbija się na dwa etapy. Instrukcja `sethi` ładuje 22 najstarsze bity, 10 najmłodszych bitów ładuje się kolejną instrukcją `or`.
	Sekwencja instrukcji
	`sethi %hi(source), destination`
	`or destination, %lo(source), destination`
	kopiuje liczbę 32-bitową ze źródła do wartości celu.
	Funkcje asemblera `%lo(const)`, `%hi(const)` zwracają odpowiednio 10 najmniej znaczących bitów i 22 najbardziej znaczące bity stałej `const`.

Tabela 30 Instrukcje przesunięcia i obrotów asemblera

OPERACJE PRZESUNIĘCIA I OBROTÓW	
INSTRUKCJA	OPIS I PRZYKŁAD
`sl[lr][x] src, number, dest`	Instrukcja wykonuje przesunięcie logiczne bitów źródła *`src`* o zadaną liczbę bitów *`number`* w lewo (`sll`) lub w prawo (`slr`) dla liczb 32 bitowych lub 64 bitowych (z sufixem x).
`sra[x] src number, dest`	Instrukcja wykonuje przesunięcie arytmetyczne bitów źródła *`src`* o zadaną liczbę bitów *`number`* w prawo dla liczb 32 bitowych lub 64 bitowych (x).

Tabela 31 Instrukcje skoków asemblera

INSTRUKCJE SKOKÓW	
INSTRUKCJA	OPIS I PRZYKŁAD
`bpcc address`	Rozgałęzienie z prognozą.
`bpr source, address`	Rozgałęzienie zależne od wartości rejestru.

`bxx label`	`a` – *branch always*, instrukcja skoku warunkowego, warunek jest zawsze prawdziwy.
	`n` – *branch never*, instrukcja skoku warunkowego, warunek zawsze jest niespełniony.
	`[n]e` – [*not*] *branch equal*, instrukcja skoku warunkowego, skok jest wykonywane, jeżeli w poprzednim porównaniu wartości były równe (`be`) lub różne (`bne`).
	Testowanie wartości pod kątem równości, wykonuje się instrukcją odejmowania z ustawieniem odpowiednich flag.
	Sekwencja instrukcji
	`subcc %r1, %r2, %g0`
	`be koniec_programu`
	przejdzie do etykiety `koniec_programu`, jeżeli wartość rejestru `%r1` równa się wartości rejestru `%r2`.
	`g[e]` – *branch greater* [*equal*], instrukcja skoku warunkowego, skok jest wykonywane jeżeli w poprzednim porównaniu druga z porównywanych wartości była większa (bg) lub większa - równa.
	Sekwencja instrukcji
	`subcc %r1, %r2, %g0`
	`bg koniec_programu`
	przejdzie do etykiety `koniec_programu`, jeżeli wartość rejestru `%r1` jest większa od wartości rejestru `%r2`.
	`l[e]` – *branch less* [*equal*], instrukcja skoku warunkowego, skok jest wykonywane jeżeli w poprzednim porównaniu druga z porównywanych wartości była mniejsza (`bl`) lub mniejsza – równa (`ble`).
	Sekwencja instrukcji
	`subcc %r1, %r2, %g0`
	`bl koniec_programu`
	przejdzie do etykiety `koniec_programu`, jeżeli wartość rejestru `%r1` jest mniejsza niż wartości rejestru `%r2`.

`call label`	Instrukcja wywołuje procedurę *label*.
`return`	Instrukcja powraca z procedury.
`save src1, src2, dest`	Instrukcja przesuwa okna rejestrów Typowym użyciem tej instrukcji jest `save %sp, -96, %sp` Rejestr R8 w oknie w procedurze wywołującej, zostanie przesunięty o 96 bitów, czyli o 23.
`restore`	Instrukcja przywraca okna rejestrów.
`prefetch fcn`	Instrukcja wstępnie pobiera dane z pamięci.

Tabela 31 Instrukcje skoków asemblera

INSTRUKCJE SKOKÓW	
INSTRUKCJA	OPIS I PRZYKŁAD
`txx number`	trap when `xx`, instrukcja przerwania systemowego o numerze `number`, przyrostek `xx` może przyjąć następujące wartości `a` – *always* – przerwanie wykonywane jest zawsze `e` – *equal* – przerwanie wykonywane jest, gdy poprzednio porównywane wartości były równe `ne` – *not equal* – przerwanie wykonywane jest, gdy poprzednio porównywane wartości były różne `g` – *greater* – przerwanie wykonywane jest, gdy z poprzednio porównywanych wartości druga była większa `ge` – *greater equal* – przerwanie wykonywane jest, gdy z poprzednio porównywanych wartości druga była większa lub równa `l` – *less* – przerwanie wykonywane jest, gdy z poprzednio porównywanych wartości druga była mniejsza `le` – *less equal* – przerwanie wykonywane jest, gdy z poprzednio porównywanych wartości druga była mniejsza. Zanim zostanie wykonana instrukcja z rodziny `txx`, należy wcześniej ustawić bity rejestru `iCC`.

Asembler procesora UltraSPARC III Cu nie został wyposażony w wiele użytecznych instrukcji. Instrukcje te nie są jednak krytycznymi cechami języka i w bardzo prosty sposób jeszcze na etapie asemblacji kodu asemblera można rozwinąć je do równoważnych sekwencji rozkazów procesora UltraSPARC III Cu. Instrukcje takie, nazywane instrukcjami symulowanymi (ang. *synthesis instructions*), zaprezentowano w **Błąd! Nie można odnaleźć źródła odsyłacza.**. Oprócz mnemonika instrukcji symulowanej znajduje się tam równoważna sekwencja pełnoprawnych instrukcji asemblera oraz opis działania instrukcji.

Tabela 32 Instrukcje symulowane

INSTRUKCJE SYMULOWANE (SYNTHETIC INSTRUCTION)		
INSTRUKCJA SYMULOWANE	RÓWNOWAŻNA SEKWENCJA INSTRUKCJI ASEBLERA	OPIS INSTRUKCJI
`set src, dest`	`sethi hi(src), dest` `or dest,%lo(src),dest`	Instrukcja ładuje liczbę 32 bitową ze źródła `src` do celu `dest`.
`clr dest`	`or %g0, %g0, dest`	Instrukcja zeruje wartość rejestru `dest`.
`cmp src1, src2`	`subcc src1, src2, %g0`	Instrukcja porównuje dwie wartości `src1 i src2`, przyrostek `cc` do instrukcji `sub` powoduje, że instrukcja nie zapisze wyniku do trzeciego operandu tylko ustawi flagi iCC.
`inc dest`	`add dest, 1, dest`	Instrukcja inkrementuje zawartość rejestru `dest`.
`dec dest`	`sub dest, 1, dest`	Instrukcja dekrementuje wartość rejestru `dest`.
`mov src, dest`	`or %g0, src, dest`	Kopiuje wartość ze źródła `src` do celu `dest`.

Tabela 32 Instrukcje symulowane

INSTRUKCJE SYMULOWANE (SYNTHETIC INSTRUCTION)		
INSTRUKCJA SYMULOWANE	RÓWNOWAŻNA SEKWENCJA INSTRUKCJI ASEBLERA	OPIS INSTRUKCJI
`not dest`	`nor dest, %g0, dest`	Neguje wartości wszystkich bitów w wartości celu `dest`. Jeżeli rejestr %r1 ma maskę bitową 0000111100001111 instrukcja not %r1 zmieni maskę maskę bitową rejestru na 1111000011110000.
`neg dest`	`sub %g0, dest, dest`	Neguje wartość najstarszego bitu w wartości celu `dest`, zmienia znak wartości na przeciwny. Jeżeli rejestr %r1 ma maskę bitową 0000111100001111 instrukcja neg %r1 zmieni maskę maskę bitową rejestru na 1000111100001111.
`nop`	`sethi 0, %g0`	Instrukcja pusta. Wykorzystywana często do wypełniania szczeliny opóźnienia.

7.6 Translacja instrukcji wysokopoziomowych na kod asemblera

Translacja wyrażeń arytmetycznych

Istnieje wiele algorytmów translacji wyrażeń matematycznych na kod asemblera. Najprostszy z nich wykorzystuje w tym celu stos.

Dla każdego węzła drzewa binarnego *W*, obliczana jest wartość każdego z jego potomków *P* i ta wartość umieszczana jest na stosie. Jeżeli obliczone są wartości wszystkich potomków *P* węzła *W*, są zdejmowane ze stosu i na nich wykonywana jest operacja właściwa dla *W*. Algorytm obliczania wyrażeń matematycznych przedstawiono na Rys. 7.8.

Jeżeli węzeł *W* jest liściem drzewa, wpisz wartość węzła do rejestru *R1*
Jeżeli węzeł *W* nie jest liściem wykonaj następujące czynności

1. Oblicz wartość lewego potomka
2. Włóż wartość lewego potomka na stos
3. Oblicz wartość prawego potomka
4. Zdejmnij wartość ze stosu do rejestru *R2*
5. Wykonaj operacje na rejestrach *R1* i *R2*.
6. Zapisz wynik operacji w rejestrze *R1*.

Rys. 7.8 Algorytm obliczania wyrażeń matematycznych w drzewie binarnym

Obliczając wyrażenia w zaprezentowany sposób, zapewnione jest, że po obliczeniu dowolnego podwyrażenia jego wynik mamy w rejestrze *R1*.

Przykład 7.1
W przykładzie zaprezentowano kod asemblera uzyskany według algorytmu z Rys. 7.8 dla instrukcji 2 + 3 * 4. Nazwy rejestrów R1 oraz R2 zamieniono na `%eax` i `%ebx`.

```
movl $2, %eax
pushl %eax
movl $3, %eax
pushl %eax
movl $4, %eax
popl %ebx
imull %ebx, %eax
popl %ebx
addl %ebx, %eax
```

□

Obliczając wyrażenia według podanego algorytmu, można zapewnić, że każdy wynik cząstkowy dowolnego podwyrażenia znajduje się z rejestrze *R1* (w tym przypadku `%eax`).

Translacja instrukcji warunkowej

Instrukcja warunkowa `if` składa się z trzech części. Pierwszą tworzy warunek zapisany jako wyrażenie, drugi blok instrukcji wykonywanych w przypadku prawdziwości warunku, trzeci opcjonalny blok instrukcji wykonywanych w przypadku nieprawdziwości warunku. Schemat instrukcji warunkowej z języka C/C++ zaprezentowano na Rys. 7.9.

```
if ( a == b ){
      /* kod wykonany, gdy warunek zwróci wartość true */
}
else{
      /* kod wykonywane, gdy warunek zwrocie wartość false */
}
```

Rys. 7.9 Przykład instrukcji warunkowej w języku C/C++

Instrukcje skoków warunkowych i bezwarunkowych oraz oznaczanie bloków przy pomocy etykiet są jedyną możliwością zapisania blokowej struktury instrukcji złożonych w tym warunkowych. Etykiety `blok_instrukcji_false` oraz `blok_nistrukcji_true` w jawny sposób zaznaczają fragmenty kodu wykonywane w przypadku odpowiednio niespełnionego i spełnionego warunku.

Instrukcja warunkowa zapisana w języku asemblera dla procesora Intel Pentium x86 oraz UltraSPARC III Cu jest równoważna sekwencji przedstawionej w Tabela 33.

Tabela 33 Zapis instrukcji warunkowej w kodzie asemblera

ASEMBLER INTEL PENTIUM x86	ASEMBLER UltraSPARC III Cu

`movl a, %eax`
`movl b, %ebx`
`cmpl %eax, %ebx`
`je blok_instrukcji_true`
`blok_instrukcji_false:`
`# kod dla czesci false`
`jmp koniec`
`blok_instrukcji_true:`
`# kod dla czesci true`
`koniec`
	`or %g0, a, %r1`
	`or %g0, b, %r1`
	`subcc %r1, %r2, %g0`
	`be blok_instrukcji_true`
	`blok_instrukcji_false:`
	`# kod dla czesci false`
	`ba koniec`
	`blok_instrukcji_true:`
	`# kod dla czesci true`
	`koniec`

Translacja instrukcji iteracyjnych

Instrukcja iteracyjna `while` składa się z warunku oraz treści pętli. Warunek sprawdzany jest przez wykonaniem ciała pętli, a ciało pętli wykonuje się dopóki warunek zwraca wartość `true`. Schemat pętli `while` zaprezentowano na **Błąd! Nie można odnaleźć źródła odsyłacza.**. Jeżeli ilość iteracji jest znana jeszcze przed wykonaniem pętli, bardziej wygodnym rozwiązaniem jest użycie pętli `for`. Pętla `for` składa się z instrukcji inicjalizacji wykonywanej przed pierwszą iteracją, instrukcji warunkowej wykonywanej przed każdą iteracją oraz instrukcji modyfikującej licznik pętli wykonywanej po każdej iteracji. Każda z tych instrukcji jest opcjonalna. Treść pętli wykonywana jest tak długo, dopóki warunek jest prawdziwy. Schemat pętli for przedstawiono w Tabela 34.

Tabela 34 Przykład instrukcji iteracyjnych, pętli `while` i `for` w języku C/C++

INSTRUKCJA PĘTLI `WHILE`	INSTRUKCJA PĘTLI `FOR`

`while ( a <= b ){`
`  /* treść pętli */`
`}`
	`for ( i = 0 ; i <= b ; ++i ){`
	`  /* treść pętli */`
	`}`

Instrukcja pętli `while` przetłumaczona na język asemblera dla procesora Intel Pentium x86 oraz UltraSPARC III Cu znajduje się w Tabela 35.

Tabela 35 Kod asemblera równoważny instrukcji interacyjnej pętli `while`

ASEMBLER INTEL PENTIUM x86	ASEMBLER UltraSPARC III Cu

`poczatek_petli:`
`  movl a, %eax`
`  movl b, %ebx`
`  cmpl %eax, %ebx`
`  jge koniec_petli:`
`tresc_petli:`
`  # kod treści pętli`
`  jmp początek_petli`
`koniec_petli:`
	`początek_petli:`
	`  or %g0, a, %r1`
	`  or %g0, b, %r1`
	`  subcc %r1, %r2, %g0`
	`  bge koniec_petli:`
	`tresc_petli:`
	`  # kod treści pętli`
	`  ba początek_petli`
	`koniec_petli:`

Instrukcja pętli `for` przetłumaczona na język asemblera dla procesora Intel Pentium x86 oraz UltraSPARC III Cu znajduje się na Tabela 36.

Tabela 36 Kod asemblera równoważny instrukcji interacyjnej pętli `for`

ASEMBLER INTEL PENTIUM x86	ASEMBLER UltraSPARC III Cu

`instrukcja_inicjalizacji:`
`  movl b, %ecx`
`poczatek_petli:`
`  # kod treści pętli`
`  decl %ecx`
`  cmpl %ecx, $0`
`  jg koniec_petli`
`  jmp poczatek_petli`
`koniec_petli:`
	`instrukcja_inicjalizacji:`
	`  or %g0, b, %r3`
	`poczatek_petli:`
	`  # kod treści pętli`
	`  sub %r3, 1, %r3`
	`  subcc %r3, 0, %g0`
	`  bg koniec_petli`
	`  ba początek_petli`
	`koniec_petli:`

Wywołanie funkcji

Funkcje są mechanizmem dostępnym w wielu językach programowania. Ich głównym celem jest rozbicie programu na mniejsze, niezależne części, których implementacja a także konserwacja jest prostsza niż jednego dużego programu. Funkcje pozwalają również wykorzystać raz napisany kod wielokrotnie.
Funkcja składa się z kilku elementów. Każda funkcja reprezentowana jest przez unikalną nazwę. W języku asemblera *nazwa funkcji* (ang. *function name*) jest adresem pierwszej instrukcji funkcji[6]. Dla ułatwienia notacji, często używa się etykiet zamiast adresów funkcji.
Parametry funkcji (ang. *function parameters*) są dane przekazywane do wnętrza funkcji ze „świata zewnętrznego" funkcji.

Funkcja może posiadać swoje własne dane, swoje własne zmienne nazywane *zmiennymi lokalnymi* (ang. *local variable*). Dane funkcji są przechowywane w jej wnętrzu i nie są dostępne na zewnątrz funkcji. Za każdym razem jak funkcja kończy swoje działanie, jej zmienne lokalne są tracone. Szczególnym rodzajem zmiennych lokalnych funkcji są jej *zmienne statyczne* (ang. *static variables*). Zmienne te nie są również dostępne poza ciałem funkcji, ale w przeciwieństwie do zmiennych lokalnych czas ich istnienia nie jest zależny od czasu wykonywania się funkcji. Zmienna statyczna jest inicjowana przy pierwszym wykonaniu funkcji, i jej czas istnienia trwa do końca działania programu.
Adres powrotu (ang. *return address*) jest niejawnym parametrem funkcji, który nie jest bezpośrednio używany w funkcji. Adres powrotu, informuje funkcję, gdzie ta ma przekazać sterowanie po zakończeniu działania. Jest to niezbędne, ponieważ jedna funkcja może być wywoływana z wielu różnych fragmentów programu i funkcja musi potrafić zwrócić sterowanie do miejsca skąd została wywołana.
Wartość zwracana (ang. *return value*) jest mechanizmem, dzięki któremu, funkcja może zwrócić efekt swojej pracy do „świata zewnętrznego" funkcji, czyli do fragmentu kodu który funkcję wywołał.

W wielu różnych językach programowania, sposób przekazywania parametr do funkcji oraz ich zwracania przez funkcję jest różny i jest nazywany konwencją wywołania (ang. *calling convention*).

Wywołania funkcji w procesorze Intel Pentium x86

Mechanizm wołania funkcji jest w architekturze Intel x86 zaimplementowany przy pomocy stosu.
Wartości na stos wkłada się przy pomocy instrukcji `pushl`, np.: `pushl $1` umieści na stosie wartość `1`. Za każdym razem, gdy na stos wkładana się wartość, stos rośnie w stronę coraz mniejszych adresów i aktualizowana jest wartość rejestru `%esp` wskazującego wierzchołek stosu. Instrukcja `pushl $1` jest równoważna sekwencji instrukcji

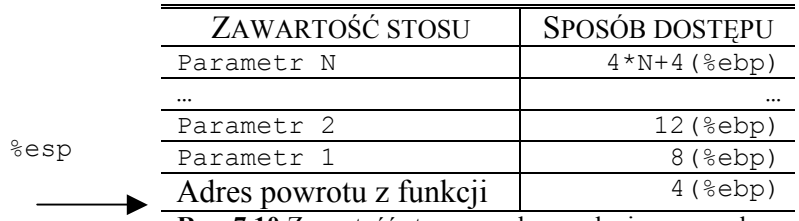
```
subl $4, %esp
movl $1, (%esp)
```

Wartość z wierzchołka stosu można pobrać instrukcją `popl`, np.: `popl %eax`. Jako parametr instrukcji `popl` podaje się, gdzie wartość ze stosu będzie zapisana. Instrukcja `popl` usuwa wartość z wierzchołka stosu. Po usunięciu wartości z wierzchołka stosu, należy zaktualizować wartość rejestru `esp`. Instrukcja `popl %eax`, jest równoważna sekwencji

[6] Również w języku C/C++ istnieje niejawna konwersja nazwy funkcji do jej wskaźnika.

```
movl (%esp), %eax[7]
add $4, %esp
```

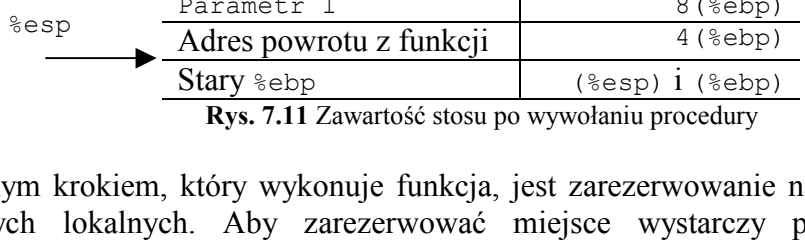
Zanim program wywoła funkcję, musi włożyć na stos parametry dla funkcji. Zgodnie w konwencja wołania języka C, parametry są wkładane na stos w kolejności odwrotnej niż ta, jaka zapisana jest w nagłówku funkcji. Następnie program wykonuje instrukcję `call` wskazując, którą funkcję ma zamiar użyć. Instrukcja `call` wykonuje niejawnie dwie rzeczy. Wkłada najpierw adres kolejnej instrukcji na stos. Jest to adres powrotu w funkcji. Gdy funkcja zakończy swoje działanie, przekaże sterowanie pod ten adres. Następnie instrukcja `call` modyfikuje zawartość rejestru `eip` (ang. *instruction pointer*) tak, żeby ten wskazywał adres pierwszej instrukcji funkcji.



ZAWARTOŚĆ STOSU	SPOSÓB DOSTĘPU
`Parametr N`	`4*N+4(%ebp)`
…	…
`Parametr 2`	`12(%ebp)`
`Parametr 1`	`8(%ebp)`
Adres powrotu z funkcji	`4(%ebp)`

Rys. 7.10 Zawartość stosu przed wywołaniem procedury

Sterowanie jest przekazane do funkcji. Pierwszą rzeczą, którą robi funkcja jest zapisanie rejestru `ebp` na stosie. Rejestr `ebp` (ang. *base pointer*) pozwala na prosty dostęp do parametrów funkcji jak również zmiennych lokalnych funkcji. Następnie funkcja kopiuje zawartość rejestru `esp` do rejestru `ebp`. W tym momencie działania programu zawartość stosu zaprezentowana jest na Rys. 7.11.



ZAWARTOŚĆ STOSU	SPOSÓB DOSTĘPU
`Parametr N`	`4*N+4(%ebp)`
…	…
`Parametr 2`	`12(%ebp)`
`Parametr 1`	`8(%ebp)`
Adres powrotu z funkcji	`4(%ebp)`
Stary `%ebp`	`(%esp)` i `(%ebp)`

Rys. 7.11 Zawartość stosu po wywołaniu procedury

Następnym krokiem, który wykonuje funkcja, jest zarezerwowanie na stosie pamięci dla zmiennych lokalnych. Aby zarezerwować miejsce wystarczy przesunąć wskaźnik wierzchołka stosu.
Jeżeli trzeba zaalokować miejsce dla zmiennej typu double, która zajmuje `8` bajtów pamięci do wskaźnika stosu dodaje się liczbę `8` `subl $8, %esp`. Zawartość stosu, kiedy funkcja zarezerwuje miejsce na stosie dla swoich zmiennych lokalnych przedstawiona jest na Rys. 7.12.

[7] Instrukcja `movl %esp, %eax` wykorzystuje adresowanie bezpośrednie, i do rejestru `eax` skopiowałaby adres wierzchołka stosu. W celu skopiowania wartości z wierzchołka stosu wskazywanego przez `esp`, należy *wyłuskać* `esp`.

ZAWARTOŚĆ STOSU	SPOSÓB DOSTĘPU
`Parametr N`	`4*N+4(%ebp)`
…	…
`Parametr 1`	`8(%ebp)`
Adres powrotu z funkcji	`4(%ebp)`
Stary `%ebp`	`(%ebp)`
Zmienna lokalna 1	`-4(%ebp)`
Zmienna lokalna 2	`-8(%ebp)` i `(%esp)`

Rys. 7.12 Zawartość stosu z zaalokowaną pamięcią za zmienne lokalne

Kiedy funkcja skończy wykonywać swoją treść musi wykonać następujące czynności. Ogólnie przyjętą konwencją, że wartość zwracana przez funkcję znajduje się w rejestrze `eax`. Funkcja powinna, więc skopiować wyliczony rezultat do rejestru `eax`. Następnie funkcja niszczy swój stos przypisując do rejestru stosu `esp` wartość rejestru bazowego. Od tego momentu wskaźnik stosu wskazuje ten sam adres w pamięci, co przed wywołaniem funkcji. Ostatnią czynnością, którą musi wykonać funkcja jest przywrócenie wartości bazowej i wykonanie instrukcji `ret`. Ponieważ podczas powrotu z funkcji, stos lokalny został zniszczony, niestateczne zmienne lokalne funkcji również przestały istnieć. Przechowywanie adresu do zmiennych lokalnych po zakończeniu działania funkcji jest błędne i może prowadzić do trudnych w wyśledzeniu błędów.

Przykład pseudokodu w asemblerze zachowania programu oraz funkcji podczas wołania funkcji pokazano na Rys. 7.13.

```
program:
      pushl parametrN-1
      pushl parametrN-2
      ...
      pushl parametr2
      pushl parametr1
      call etykieta_funkcji

etykieta_funkcji:
      pushl %ebp              # zapamiętaj wartość rejestru ebp
      movl %esp, %ebp         # przypisz wskaźnik stos
                              # wskaźnika bazowego
                              # zarezerwuj miejsce dla zmiennych
                              # lokalnych
      subl $suma_rozmiarow_zmiennych_lokalnych, %esp
      ...          # treść funkcji
                   # skopiuj wartość zwracaną przez funkcję do eax
      movl wartość_zwracana, %eax
      movl %ebp, %esp         # przywróć wskaźnik stosu
      popl %ebp               # przywróć wskaźnik bazowy
      ret
```

Rys. 7.13 Schemat wywołania funkcji w asemblerze dla architektury Intel

```
#Program oblicza wyrażenie 2^3 + 5^2 korzystając z funkcji `power'
.section .data
.section .text
.globl _start
_start:
pushl $3                    # włóż pierwszy argument
pushl $2                    # włóż drugi argument
call power              # wywołaj funkcję
addl $8, %esp               # przywróć wskaźnik stosu
pushl %eax              # zachowaj pierwszy wynik cząstkowy
pushl $2                    # włóż pierwszy argument
pushl $5                    # włóż drugi argument
call power              # wywołaj funkcję
addl $8, %esp               # przywróć wskaźnik stosu
popl %ebx                   # drugi wynik cząstkowy


addl %eax, %ebx
movl $1, %eax
int $0x80

#Oblicza potęgę liczb całkowitych
#INPUT:
# pierwszy argument – podstawa potęgi
# Drugi argument – wykładnik
#OUTPUT:
# pierwszy argument ^ drugi argument
#VARIABLES:
# %ebx – przechowuje podstawę potęgi
# %ecx – przechowuje wykładnik

.type power, @function
power:
pushl %ebp                  # zachowaj wskaźnik bazowy
movl %esp, %ebp             # przypisz wsk. stosu do wsk. bazowego
subl $4, %esp               # zarezerwuj miejsce na zmienne lokalne
movl 8(%ebp), %ebx          # zapisz pierwszy argument w %eax
movl 12(%ebp), %ecx         # zapisz drugi argument w %ecx
movl %ebx, -4(%ebp)         # zapisz wynik pośredni
power_loop_start:
      cmpl $1, %ecx         # jeżeli wykładnik jest równy jeden
                            # zakończ
      je end_power
      movl -4(%ebp), %eax   # zapisz wynik pośredni do %eax
      imull %ebx, %eax      # pomnóż ebx przez eax, zapisz w eax
      movl %eax, -4(%ebp)   # zachowaj aktualny wynik
      decl %ecx             # zdekrementuj wykładnik
      jmp power_loop_start  # skocz do etykiety `power_loop_start'
      end_power:
      movl -4(%ebp), %eax         # zapisz wynik w %eax
      movl %ebp, %esp             # przywróć wskaźnik stosu
      popl %ebp                   # przywróć wskaźnik bazowy
      ret
```

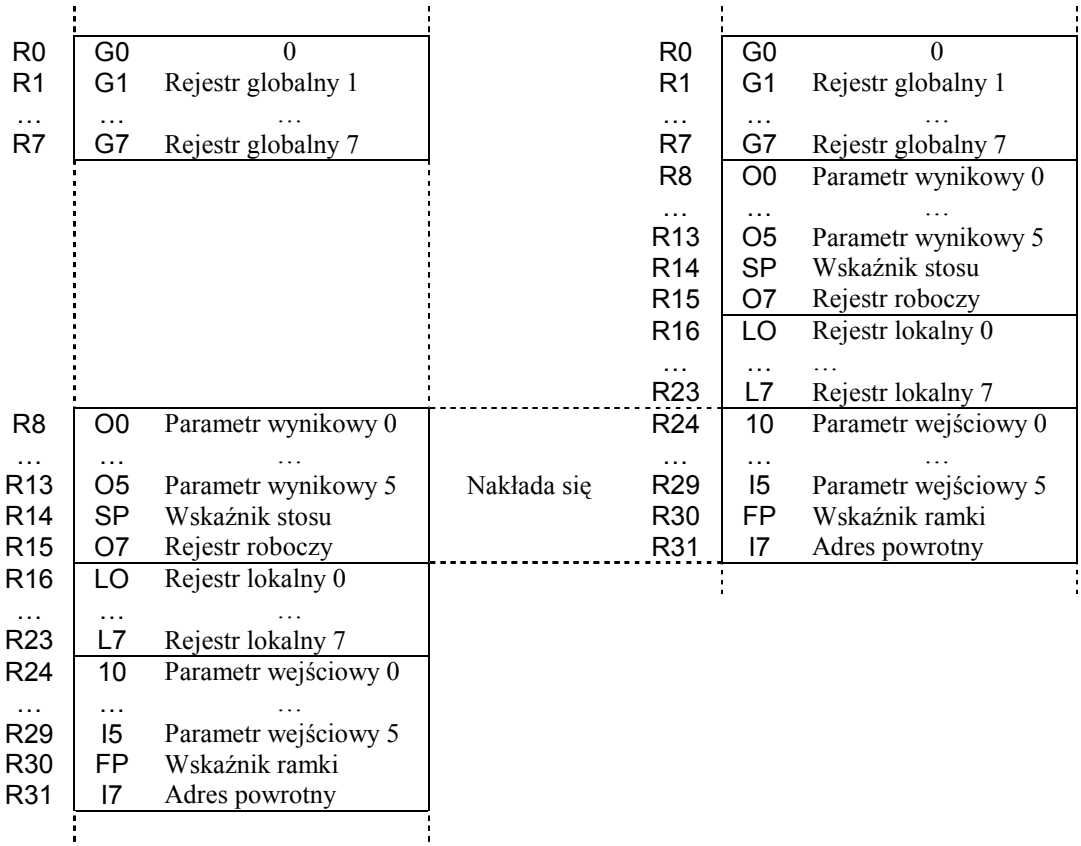
Rys. 7.14 Pełny kod programu w języku asembler dla architektury Intel x86

Wywołania funkcji w procesorze UltraSPARC III Cu

Wywołanie procedury w procesorze Intel x86 wymaga przynajmniej czterech odwołań do pamięci: włożenie adresu powrotu na stos, włożenie wskaźnika bazowego na stos, zdjęcie

wskaźnika bazowego ze stosu[8], zdjęcie adresu powrotu ze stosu oraz po dwa odwołania do pamięci dla każdego rejestru, w celu zachowania oraz przywrócenia jego wartości na taką, jaką miał przed wywołaniem procedury.

Architektura procesora UltraSPARC III Cu stara się wyeliminować tak wiele odwołań do pamięci jak to tylko możliwe. Układ wyposażony został w okna pamięci oraz dwie instrukcje `save` i `restore`. W danej chwili widoczne jest tylko jedno okno wskazywane przez rejestr CWP. Zadaniem okna jest emulowanie ramek na stosie już wewnątrz układu procesora.



Rys. 7.15 Nakładanie się okien rejestrów w procesorze UltraSPARC III Cu

Podczas wywołania funkcji, okno rejestrów jest przesuwane w ten sposób, że rejestry wejściowe procedury wywoływanej oznaczone od R24 do R31 nakładana są na rejestry wyjściowe procedury wywołującej oznaczone od O0 do O7 nakładają się.
Rejestr R15 w procedurze wywołującej przesłaniany jest przez rejestr R31 procedury wywoływanej ma szczególną rolę. Przechowuje on adres powrotny.

W celu uzyskania przez program pamięci na zmienne lokalne w ciele procedury, wystarczy tylko przesunąć wskaźnik wierzchołka stosu. Ponieważ stos zaczyna się w wysokiej pamięci i rośnie w stronę coraz niższych adresów rezerwowanie miejsca na stosie polega na odjęciu od jego wierzchołka liczby bajtów potrzebnego miejsca.

[8] Czynności te są wykonywane niejawnie przez instrukcje `call` i `ret`

Instrukcja `sub %sp, 60, %sp` rezerwuje 60 bajty na stosie. Jest to jednak instrukcja błędna.

W procesorze UltraSPARC III Cu w celu zwiększenia efektywności architektury, cała pamięć jest wyrównana. Oznacza to, że dwubajtowa zmienna może być ulokowana tylko pod adresem podzielnym przez dwa, czterobajtowe słowo tylko pod adresem podzielnym przez cztery itd.

Z tego powodu stos jest zawsze wyrównany do podwójnego słowa czyli ośmiu bajtów. Żaby zapewnić, że stos jest zawsze oprawnie wyrównany wskaźnik wierzchołka stosu musi być podzielny przez osiem. Jeżeli użytkownik potrzebuje 60 bajtów pamięci, musi zarezerwować 64 żeby utrzymać stos wyrównany.

Prostym sposobem zapewnienia, że liczba jest podzielna przez osiem, jest wyzerowanie trzech najmniej znaczących bitów z zapisie dwójkowym. Przykład liczb całkowitych oraz ich odpowiedników z wyczyszczonymi trzema najmniej znaczącymi bitami pokazano w Tabela 37.

Tabela 37 Obcięte liczby całkowite podzielne przez osiem

ZAPIS DZIESIĘTNY	ZAPIS BINARNY	WARTOŚĆ UCIĘTA	WYNIK
17	010001	010000	16
16	010000	010000	16
15	001111	001000	8
9	001001	001000	8
8	001000	001000	8
7	000111	000000	0
1	000001	000000	0
-1	111111	111000	-8
-7	111001	111000	-8
-8	111000	111000	-8
-9	110111	110000	-16

Wyzerowanie trzech najmłodszych bitów powoduje uzyskanie wartości podzielnej przez osiem, lecz jest to wartość mniejsza lub równa od wartości pierwotnej. Jeżeli użytkownik potrzebuje zarezerwować na stosie 60 (`111100b`) bajtów pamięci, to po wyzerowaniu trzech najmłodszych bitów uzyska on 56 (`111000b`) bajty pamięci. Jest to zjawisko niepożądane dlatego należy posłużyć się bardziej zaawansowaną arytmetyką liczb w notacji dwójkowej.
Zamiast odejmować od stosu dodatnią liczbę bajtów pamięci do zaalokowania, można do wierzchołku stosu dodać ujemną liczbę bajtów pamięci. Uzasadnieniem takiej arytmetyki jest to, że dla liczb ujemnych, czyszcząc ich trzy najmłodsze bity uzyskujemy wartość większą co do wartości bezwzględnej niż pierwotna liczba. Przykład wyrównanych wartości bezwzględnych dla liczb dodatnich i ujemnych pokazano w Tabela 38.

Tabela 38 Wartość bezwzględna uciętej wartości dla liczb dodatnich i ujemnych

ZAPIS DZIESIĘTNY	ZAPIS BINARNY	WARTOŚĆ UCIĘTA	WYNIK	| WYNIK |
60	111100	111000	56	56
9	001001	001000	8	8
-9	110111	110000	-16	16
-60	1000100	1000000	-64	64

Instrukcja która poprawnie alokuje miejsce na stosie dla zmiennych lokalnych ma postać

```
add %sp, -60 & 0xffffff8, %sp
```

a ponieważ `0xffffff8` jest zapisem liczby –8, można napisać

```
add %sp, -60 & -8, %sp
```

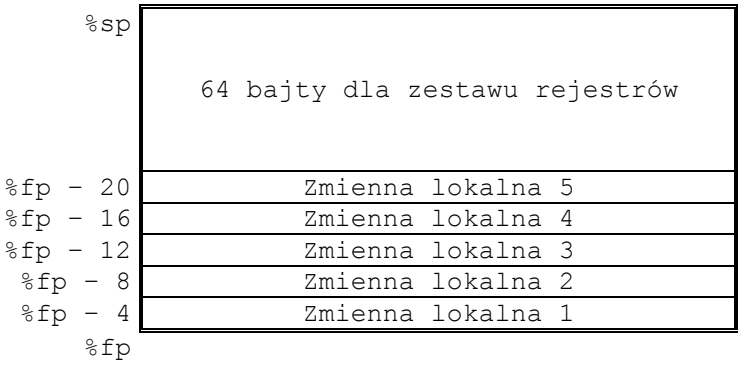
Podczas wywołania funkcji, funkcja wywoływana musi zapamiętać zestaw rejestrów oraz zarezerwować miejsce dla zmiennych lokalnych. Wykonuje to instrukcją

```
save %sp, -64 – pamięć_dla_zmiennych_lokalnych, %sp
```

Jeżeli funkcja potrzebuje zarezerwować miejsce dla pięciu zmiennych o rozmiarze jednego słowa, wykona instrukcję

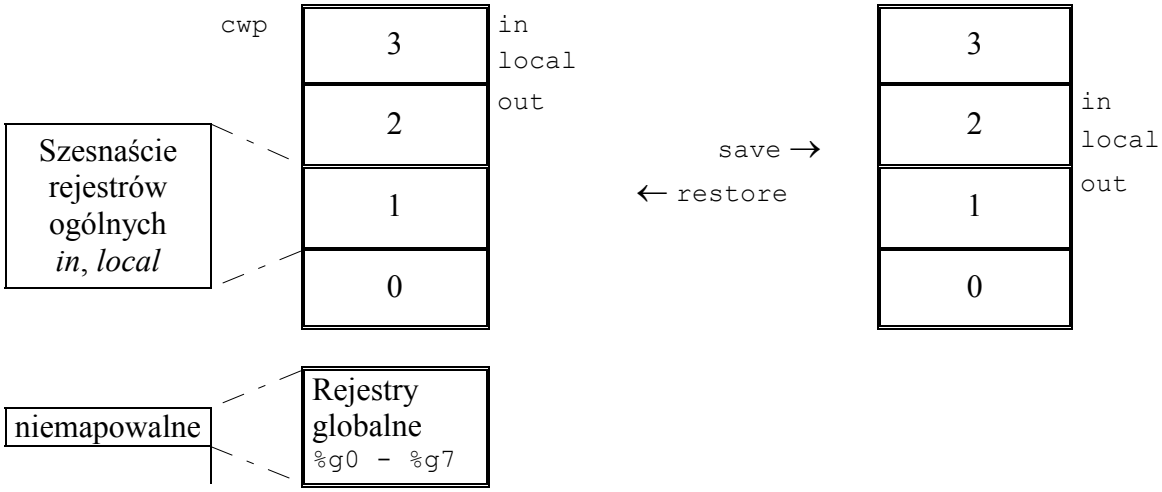
```
save %sp, ( -64 – ( 5 * 4 ) ) & -8, %sp
```

Zawartość stosu za zarezerwowaniu miejsca na szesnaście czterobajtowych rejestrów oraz pięć czterobajtowych zmiennych zaprezentowano na Rys. 7.16.



Rys. 7.16 Zawartość stosu po zaalokowaniu miejsca dla zestawu rejestrów i zmiennych lokalnych

Liczba 64 bajtów niezbędnych do przechowania rejestrów wynika z budowy okna rejestrów. Każde okno rejestrów zawiera 8 rejestrów grupy *in* i 8 rejestrów grupy *local*. Rejestry grupy *out* nakładają się na rejestry grupy *in* kolejnego okna, dlatego nie przechowuje się ich w oknie rejestrów. Te grupy rejestrów są mapowane (ang. *mapping*).



Rys. 7.17 Okna rejestrów

Rejestry z grupy *general* odpowiadają zmiennym globalnym i w całym systemie przechowywana jest tylko jedna ich kopia. Mówi się, że ta grupa rejestrów nie jest mapowana. Okno rejestrów przedstawione jest na Rys. 7.17.

Na Rys. 7.18 przedstawiono pełen program obliczający silnię zapisany w języku asemblera dla plaftormy SPARC. Zaprezentowany kod nie jest zoptymalizowany.

```
.file   "silnia_1.c"
gcc2_compiled.:
        .global .umul
.section        ".text"
        .align 4
        .global silnia
        .type    silnia,#function
        .proc   04
silnia:
        !#PROLOGUE# 0
        save    %sp, -120, %sp
        !#PROLOGUE# 1
        st      %i0, [%fp+68]   ! zachowanie do pamięci wartości n
        mov     1, %o0          ! przesunięcie do rejestru 1
        st      %o0, [%fp-24]   ! zapisanie tej wartości do s
        mov     1, %o0          ! przesunięcie do rejestru 1
        st      %o0, [%fp-20]   ! zapisanie tej wartości do i
.LL3:
        ld      [%fp-20], %o0   ! pobranie wartości i
        ld      [%fp+68], %o1   ! pobranie wartości n
        cmp     %o0, %o1        ! porównanie (wartości i oraz n)
        ble     .LL6            ! skok warunkowy (jeśli i < n)
        nop                     ! instrukcja pusta
        b       .LL4            ! skok bezwarunkowy
         nop                    ! instrukcja pusta
.LL6:
        ld      [%fp-24], %o0   ! pobranie wartości s
        ld      [%fp-20], %o1   ! pobranie wartości i
        call    .umul, 0        ! wywołanie funkcji mnożenia
         nop                    ! isnstrukcja pusta
        st      %o0, [%fp-24]   ! przepisanie wyniku do s
.LL5:
        ld      [%fp-20], %o0   ! pobranie wartości i
        add     %o0, 1, %o1     ! dodanie 1 do i
        st      %o1, [%fp-20]   ! zapisanie wartości i
        b       .LL3            ! skok na początek pętli
         nop                    ! instrukcja pusta
.LL4:
        ld      [%fp-24], %o0   ! pobranie wartości s
        mov     %o0, %i0        ! przesunięcie wartości
        b       .LL2
         nop
.LL2:
        ret
        restore
.LLfe1:
        .size    silnia,.LLfe1-silnia
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

Rys. 7.18 Program obliczający silnie zapisany w kodzie asemblera dla platformy SPARC [28]

8 Kompilator *mtc* (*Master Thesis Compiler*)

W rozdziale przedstawiono instalację, użycie oraz wewnętrzną budowę kompilatora *mtc*.

8.1 Instalacja programu

Aplikacja dołączona do pracy dyplomowej, w całości napisana jest w języku C++. Do skompilowania kodu wymagane są: biblioteka boost [21], biblioteka Loki [25], zewnętrzne narzędzie flex++ [23] oraz zewnętrzne narzędzie bison++ [20].

Flex++ jest generatorem analizatorów leksykalnych. Wczytuje plik wejściowy który zawiera wyrażenia regularne oraz kod języka C/C++. Jako wynik, generuje plik w pełni w języku C/C++, który zawiera kod analizatora leksykalnego. Reguły leksykalne dla narzędzia *flex*++ są zawarte w pliku *scanner.l*.

Bison++ jest generatorem parserów dla gramatyk LR(1). Program wczytuje plik z regułami gramatycznymi oraz kodem języka C/C++. Jako wynik generuje kod w pełni w języku C/C++ parsera. *Bison*++ jest zaprojektowany do współpracy ze skanerem wygenerowanym przez *flex*++. Opis gramatyki języka zawarty jest w pliku *bison_parser.y*.

Kompilacja kodu polega na wykonaniu komendy *make all*. Domyślnym trybem kompilacji jest kompilacja RELEASE. Podczas niej wykonywane są optymalizacje, a poziom wyświetlania ostrzeżeń ustawiony jest na minimalny. Istnieje możliwość zamiany trybu kompilacji na DEBUG w pliku *Makefile*.

Wynikiem kompilacji kodu źródłowego jest aplikacja `mtc`.

8.2 Używanie aplikacji

Aplikacja pracuje w trybie wsadowym. W celu generowania kodu asemblera z pliku `source.mtc`, należy użyć polecenia `./mtc source.mtc`.

W przypadku wykrycia w kodzie błędów, użytkownik zostanie o tym powiadomiony komunikatem a dalszy proces generowania asemblera zostanie przerwany. Jeżeli kod użytkownika będzie bezbłędny aplikacja wydrukuje na standardowe wyjście kod asemblera.

W celu utworzenia pliku wynikowego gotowego do uruchomienia, należy posłużyć się dołączonym skryptem shellowym `mtc.sh` podając nazwę pliku z kodem bez rozszerzenia. W celu wygenerowania pliku wykonywalnego z pliku `source.mtc` należy użyć polecenia `./mtc.sh source`.

Proces tworzenia aplikacji wykonywalnej jest kilkustopniowy. Pierwszym etapem jest generowanie kodu asemblera według polecenia

```
./mtc $1".mtc" > $1".s"
```

Następnie kod asemblera jest przetwarzany na plik obiektowy poleceniem

```
as $1".s" -o $1".o"
```

Plik obiektowy jest linkowany do aplikacji wykonywalnej komendą

```
ld $1".o" -o $1.
```

8.3 Opis symboli leksykalnych

Pierwszym etapem analizy pliku użytkownika jest wczytanie go przez skaner. Skaner rozpoznaje w nim identyfikatory, operatory oraz liczby. Usuwa również jednoliniowe komentarze.

W powodów wydajnościowych, analizator leksykalny nie rozpoznaje słów kluczowych. Jeżeli dopasowana zostaje sekwencja **litera** * (**litera** | **cyfra**), analizator zwraca ją do modułu `KeywordsTable` który decyduje o tym, czy rozpoznany napis jest słowem kluczowym czy jest identyfikatorem. Dzięki temu automat stanowy skanera jest mniej złożony.

Obsługiwane są następujące słowa kluczowe: `do`, `else`, `for`, `if`, `return`, `while` oraz typy danych `int` i `void`.

Wyrażenie regularne oraz odpowiadające im symbole leksykalne zaprezentowano w Tabela 39.

Tabela 39 Wyrażenia regularne i odpowiadające im symbole terminalne

WYRAŻENIE REGULARNE	SYMBOL TERMINALNY
[\t]	**whiteChar**
\n	**newLine**
[_a-zA-Z]	**letter**
[0-9]	**digit**
`{letter}({letter}|{digit})*`	**identifier**
`\"[^"\n]*\"`	**string**
`{digit}{1,4}`	**integer**

8.4 Opis składni języka

Sekwencja symboli terminalnych rozpoznanych przez skaner jest składana w drzewo syntaktyczne przez parser. Parser buduje drzewo na podstawie produkcji tworzących gramatykę.

Program użytkownika składa się z sekwencji funkcji. Wymagana jest co najmniej jedna funkcja. Funkcja może zawierać wyrażenie matematyczne jak również instrukcje sterujące.

Obsługiwane są następujące instrukcje sterujące:

- pętla `for`
- pętla `while`
- pętla `do while`
- instrukcja warunkowa `if, else`.

Funkcja zwraca wartość wyrażenia występującego po słowie kluczowym `return`.

Gramatyka języka akceptowalnego przez kompilator przedstawiona jest na Rys. 8.1 oraz Rys. 8.2.

```
program:
      functions_list

functions_list:
        functions_list function
      | function

function:
      identifier identifier '(' paramaters_list ')' instruction

paramaters_list:
        paramaters_list ',' declaration
      | declaration
      | /*empty*/

declaration:
        identifier identifier
      | identifier identifier '=' expression

instruction:
        compound_instruction
      | if_instruction
      | while_instruction
      | for_instruction
      | return_instruction
      | declaration ';'
      | assignment_expression ';'

compound_instruction:
      '{' instructions_list '}'

instructions_list:
        instructions_list instruction
      | instruction

if_instruction:
        IF '(' assignment_expression ')' instruction %prec IFX
      | IF '(' assignment_expression ')' instruction
            ELSE instruction

while_instruction:
      WHILE '(' assignment_expression ')' instruction

for_instruction:
      FOR '(' declaration ';'
      assignment_expression ';'
      assignment_expression ')' instruction
```

Rys. 8.1 Gramatyka języka akceptowanego przez kompilator

```
do_while_instruction:
      DO '{' instruction '}' WHILE '(' assignment_expression ')'';'

return_instruction:
        RETURN ';'
      | RETURN assignment_expression ';'

assignment_expression:
        assignment_expression '=' expression
      | expression

expression:
       expression '<' simple_expression
      | expression '>' simple_expression
      | expression EQ simple_expression
      | expression NEQ simple_expression
      | expression OR simple_expression
      | expression AND simple_expression
      | simple_expression

simple_expression:
        simple_expression '+' factor
      | simple_expression '-' factor
      | factor

factor:
        factor '*' unary_factor
      | factor '/' unary_factor
      | unary_factor

unary_factor:
        '+' unit
      | '-' unit
      | unit

unit:
        identifier
      | function_call
      | NUMBER
      | STRING
      | parenthesis

function_call:
      identifier '(' expressions_list ')'

expressions_list:
        expressions_list ',' assignment_expression
      | assignment_expression
      | /* NULL */

parenthesis:
      '(' assignment_expression ')'

identifier:
      IDENTIFIER
```

Rys. 8.2 Gramatyka języka akceptowanego przez kompilator

8.5 Opis reguł semantycznych

Reguły semantyczne opisują znaczenie instrukcji składniowych.

Przykład 8.1
Reguły składniowe opisują pętle `for` jako

```
for ( deklaracja ; wyrażenie ; instrukcja ) instrukcja
```

ale nic nie mówią o kolejności wykonywania deklaracji, wyrażenia oraz instrukcji. Opisują to reguły semantyczne języka.□

Program użytkownika składa się funkcji. Wymagana jest przynajmniej jedna funkcja o nazwie `main`, od której zacznie się wykonywanie programu. Jeżeli funkcja `main` nie zostanie odnaleziona, użytkownik zostanie o tym powiadomiony komunikatem błędu a proces kompilacji będzie przerwany. Funkcja przyjmuje dowolną ilość parametrów, zwraca opcjonalnie jedną wartość. Mechanizm przekazywania argumentów do funkcji jest mechanizmem *przez wartość*. Nazwy funkcji muszą mieć unikalne nazwy w całej jednostce translacji.

Przykład 8.2
Najprostszy poprawny program akceptowany przez kompilator `mtc` składa się z funkcji `main` zwracającej wartość całkowitoliczbową.

```
int main(){
      return 0;
}
```
□

Program obsługuje wyrażenia matematyczne na typie całkowitym. Dostępne są operatory matematyczne dwuargumentowe: dodawania +, odejmowania –, mnożenia *, dzielenia /, operatory matematyczne jednoargumentowe: plus +, minus –, operatory logiczne: równy ==, różny !=, mniejszy <, większy >, alternatywa &&, koniunkcja || oraz operator nawiasu ( ). Wyrażenia są obliczane z zachowaniem priorytetu operatorów.

Przykład 8.3
Przykład wyrażeń matematycznych obsługiwanych przez kompilator `mtc`

```
int main() {
      return 2 + 3*4 - 5 == ( 2!=3 ) + 1 + 3*4 - ( 4+1 );
}
```
□

W wewnętrznym języku istnieją dwa typy atomowe. Typ `int` jest pełnoprawnym typem. Można tworzyć zmienne tego typu i używać je w wyrażeniach matematycznych. Zmienna lokalna może być zainicjowana natychmiast po jej utworzeniu opcjonalną instrukcją inicjalizacji. Zmienne lokalne niezainicjowane przechowują wartości losowe, a o ich wystąpieniu użytkownik powiadamiany jest komunikatami ostrzegawczymi. Literał liczbowy jest typu `int`.
Typ `void` służy jedynie do zasygnalizowania, że funkcja nie zwraca żadnej wartości. Nie jest możliwe zadeklarowanie zmiennej typu `void`.

Wyrażenia logiczne zwracają wartość całkowitoliczbową równą 1 (`true`) lub 0 (`false`).

Przykład 8.4
Dozwolone jest deklarowanie zmiennych typu `int`. Typ `void` może być zwracany tylko przez funkcje.

```
// funkcja która nie powinna zwracać wartości, ma typ void
void main() {
      int liczba = 3;
      // void zmienna – ta instrukcja jest błędna
}
```

W ciele funkcji można deklarować zmienne lokalne typu `int`. Dwie zmienne o takiej samej nazwie zdefiniowane w dwóch różnych funkcjach traktowane są jakie dwa różne symbole. Dwie zmienne o takiej samej nazwie zdefiniowane w jednej funkcji, pomimo różnych zasięgów lokalnych traktowane są jako błędnie zdefiniowane. Po napotkaniu drugiej takiej definicji, użytkownik zostanie powiadomiony o tym komunikatem błędu a dalszy proces kompilacji zostaje przerywany.

Przykład 8.5
Przykład poprawnego programu dla kompilatora `mtc` z wykorzystaniem zmiennych lokalnych.

```
int funkcja( int _liczba ) {
      // niepoprawne, _liczba to parametr funkcji
      // int _liczba;

      int liczba = 3;

      if ( liczba == 3 ) {
            // liczba zostala juz zdefiniowana w ciele funkcji
            // int liczba = 0;
      }
}

// _liczba w funkcji main i _liczba w funkcji funkcja
// to dwa różne symbole
int main( int _liczba ) {
      return _liczba;
}
```

Wywołanie funkcji sprowadza się do podania nazwy funkcji wraz z argumentami otoczonymi nawiasem. Argumentem funkcji może być dowolne wyrażenie, w tym wartość zwracana przez inną funkcję. Wywołania rekurencyjne funkcji są poprawną konstrukcją języka.
Podczas rekurencyjnego wywoływania funkcji, należy zapewnić, że każda wołana instancja funkcji posiada swój własny stos. Dzięki temu zmienne lokalne oraz parametry nie są nadpisywane.

W przypadku, gdy liczba parametrów i argumentów wywołania funkcji, lub ich typy są niezgodne, użytkownik zostanie o tym powiadomiony komunikatem błędu, a dalszy proces kompilacji zostaje wstrzymany.

Przykład 8.6
Przykład poprawnego programu dla kompilatora `mtc` z wykorzystaniem funkcji.

```
int silnia( int _liczba ) {
      if ( _liczba == 0 ) {
            return 1;
      }
      else {
            return _liczba * silnia( _liczba - 1 );
      }
}

int main( ) {
      return silnia( 10 - 2 + 2 * 2 );
}
```

W języku dla kompilatora `mtc` dostępne są następujące instrukcje sterujące:
- pętla `for`
- pętle `while`
- warunek `if`, `else`

Zarówno pętla `for` jak i `while` wykonują się dopóki warunek logiczny jest spełniony. W obu przypadkach, warunek sprawdzany jest przed wykonaniem pętli. Jest zatem możliwe, że treść pętli nie wykona się ani raz. Użycie pętli `for` jest bardziej wygodne w przypadku, jeżeli liczba iteracji jest znana przed wykonaniem pętli.
Instrukcja warunkowa `if` umożliwia zmianę sekwencji działania programu w zależności od pewnych czynników, np.: danych podanych przez użytkownika. Część wykonywana w przypadku niespełnienia warunku logicznego jest opcjonalna i użytkownik nie musi jej specyfikować.

Przykład 8.7
Przykład poprawnego programu dla kompilatora `mtc` z wykorzystaniem instrukcji sterujących `while`, `for` oraz `if`.

```
int silnia_while ( int _liczba ) {
      int wynik = 1;
      while ( _liczba != 0 ) {
            wynik = wynik * _liczba;
            _liczba = _liczba - 1;
      }
      return wynik;
}

int silnia_for ( int _liczba ) {
      int wynik = 1;
      for ( int wynik = 1 ; _liczba != 0 ; _liczba = _liczba - 1 ){
            wynik = wynik * _liczba;
      }
}

int main( ) {
      return silnia_while( 5 ) - silnia_for( 3 );
}
```

8.6 Budowa aplikacji

Aplikacja `mtc` została napisana z wykorzystaniem technik obiektowych. Problem kompilacji został rozbity na wiele abstrakcji a każda z nich reprezentowana jest przez jedną lub zbiór klas. Dołożono starań, żeby jedna klasa miała dokładnie jedno zadanie.

Schemat ogólny

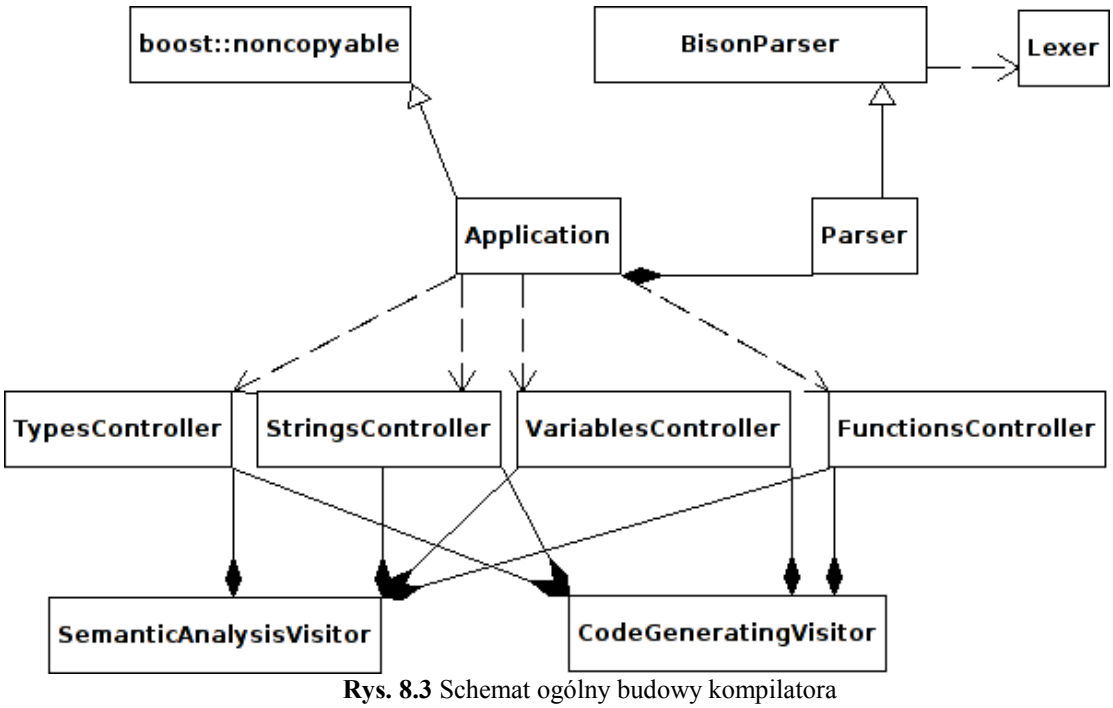
Główną klasą aplikacji jest `Application`. Do metody `execute` przekazuje się ilość argumentów wiersza poleceń oraz te argumenty. Jeżeli kompilacja powiedzie się, funkcja zwraca wartość 0.

Klasy `Lexer` jest klasą wygenerowaną przez program flex++. Odpowiada ona za skanowanie pliku użytkownika. Metoda `yylex`() zwraca symbol leksykalny , który udało się odnaleźć. Metoda `yyText`() zwraca napis skojarzony z tym symbolem.

Klasa `BisonParser` jest klasą wygenerowaną przez program `bison`++. W celu dostosowania interfejsu klasy `BisonParser` do potrzeb aplikacji użyto wzorca projektowego *Adapter* w postaci klasy `Parser`. Klasa `Parser` pozwala w prosty sposób przekazać strumień wejściowy, skąd czytane będzie plik do analizy, oraz strumień wyjściowy, gdzie zwracane będą komunikaty błędów.

Analizę semantyczną zbudowanego drzewa składniowego wykonuje klasa `SemanticAnalysisVisitor`. W celu uproszczenia kodu, jego możliwości rozbudowy jak również zmiany funkcjonalności, kod który analizuje drzewo semantycznie zebrano w jednym miejscu jako wzorzec projektowy W*izytator*.

Informacje potrzebne go analizy semantycznej oraz do wygenerowania kodu asemblera są zbierane przez klasę `SemanticAnalysisVisitor` w kontenerach: `TypesController`, `StringsController`, `VariablesController` oraz `FunctionsController`.



Rys. 8.3 Schemat ogólny budowy kompilatora

Podobnie jak analiza semantyczna, generowanie kodu asemblera wykonywane jest przez klasę wizytatora, `CodeGeneratingVisitor`. Klasa ta korzysta z informacji zebranych przez klasę `SemanticAnalysisVisitor`. Odpowiedzialna za to jest klasa `CodeGenerator`. Klasa `CodeGeneratingVisitor` odpowiada za generowanie kodu do obsługi stosu, wyrażeń matematycznych, zmiennych lokalnych. Zawiera ona informacje o sposobie generowaniu asemblera, nie posiada natomiast wiedzy o konkretnych literałach tekstowych używanych w asemblerze. Znajomość składni asemblera (w tym przypadku AT&T) została przeniesiona do klasy `CodeGenerator`.

Drzewo składniowe

Klasy reprezentujące elementy gramatyki zostały zamknięte w przestrzeni nazw `Syntax`. Każdy symbol gramatyki jest w programie odwzorowany jako jedna klasa. Wszystkie klasy drzewa składniowego są pochodne klasy `ISyntaxTreeNode`. Klasa `ISyntaxTreeNode` dziedziczy po `boost::noncopyable`, zabezpieczając przed kopiowaniem obiektów, oraz po `Loki::BaseVisitable` tworząc interfejs pozwalający klasie `CodeGeneratingVisitor` na „odwiedzanie" klas drzewa składniowego.

Konstruktor tych klas, pozwalają tworzyć drzewo składniowe według produkcji gramatycznych.

Przykład 8.8
Dla produkcji gramatycznej

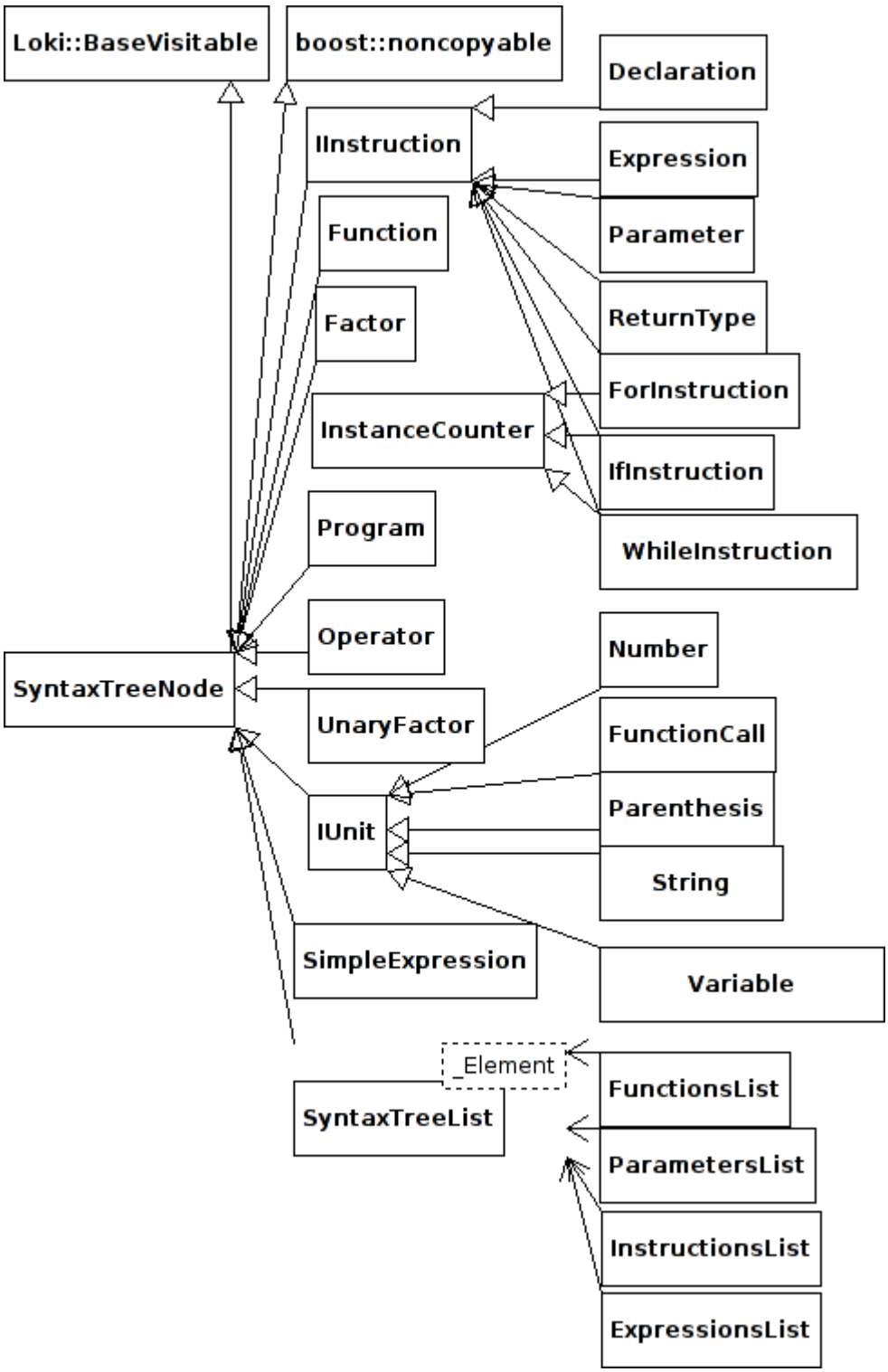
```
assignment_expression:
        assignment_expression '=' expression
      | expression
```

klasa `AssignmentExpression` posiada dwa konstruktory. Pierwszy pozwala utworzyć obiekt `AssignmentExpression` z obiektu `AssignmentExpression` i obiektu `Expression`, drugi tylko z obiektu `Expression`.

```
AssignmentExpression::AssignmentExpression (
      AssignmentExpression,
      Expression
);

AssignmentExpression::AssignmentExpression (
      Expression
);
```

□

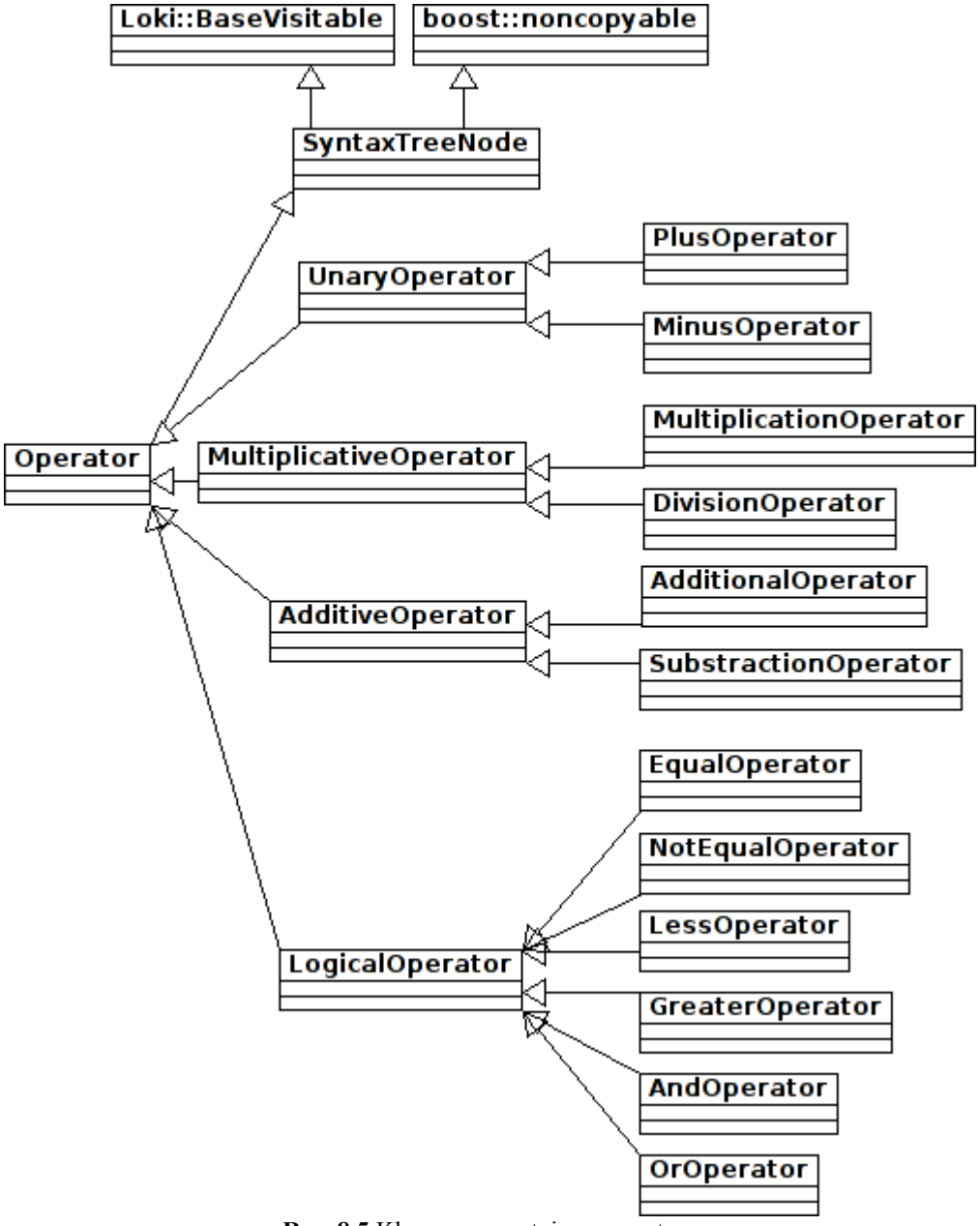
W celu generowania unikalnych etykiet dla pętli do `while`, `for`, `while` oraz instrukcji warunkowej `if else`, zliczana jest liczba utworzonych instancji klas `DoWhileInstruction` , `ForInstruction`, `WhileInstruction`, `IfInstruction`. Wspólny kod zliczania instancji, został przeniesiony do klasy bazowej `InstanceCounter`. W celu zliczania liczby utworzonych elementów klas `DoWhileInstruction`, `ForInstruction`, `WhileInstruction` i `IfInstruction` każdej z osobna, przyjęto wzorzec *Curiously Reccuring Template* [19].



Rys. 8.4 Klasy reprezentujące symbole gramatyki

Symbole gramatyki reprezentujące konstrukcje lewostronnie rekurencyjne *(A → A b | b)* takie jak `FunctionsList`, `ParametersList`, `InstructionsList`, `ExpressionsList` zostały wyprowadzone jako konkretyzacje szablonu `ISyntaxTreeList`.

Operatory wspierane przez język kompilatora `mtc` zgrupowane są w hierarchię klas, której korzeniem jest klasa `Operator`. Klasa ta dziedziczy po `ISyntaxTreeNode`, blokując kopiowanie oraz dostarczając interfejs dla wzorca *Wizytator*.

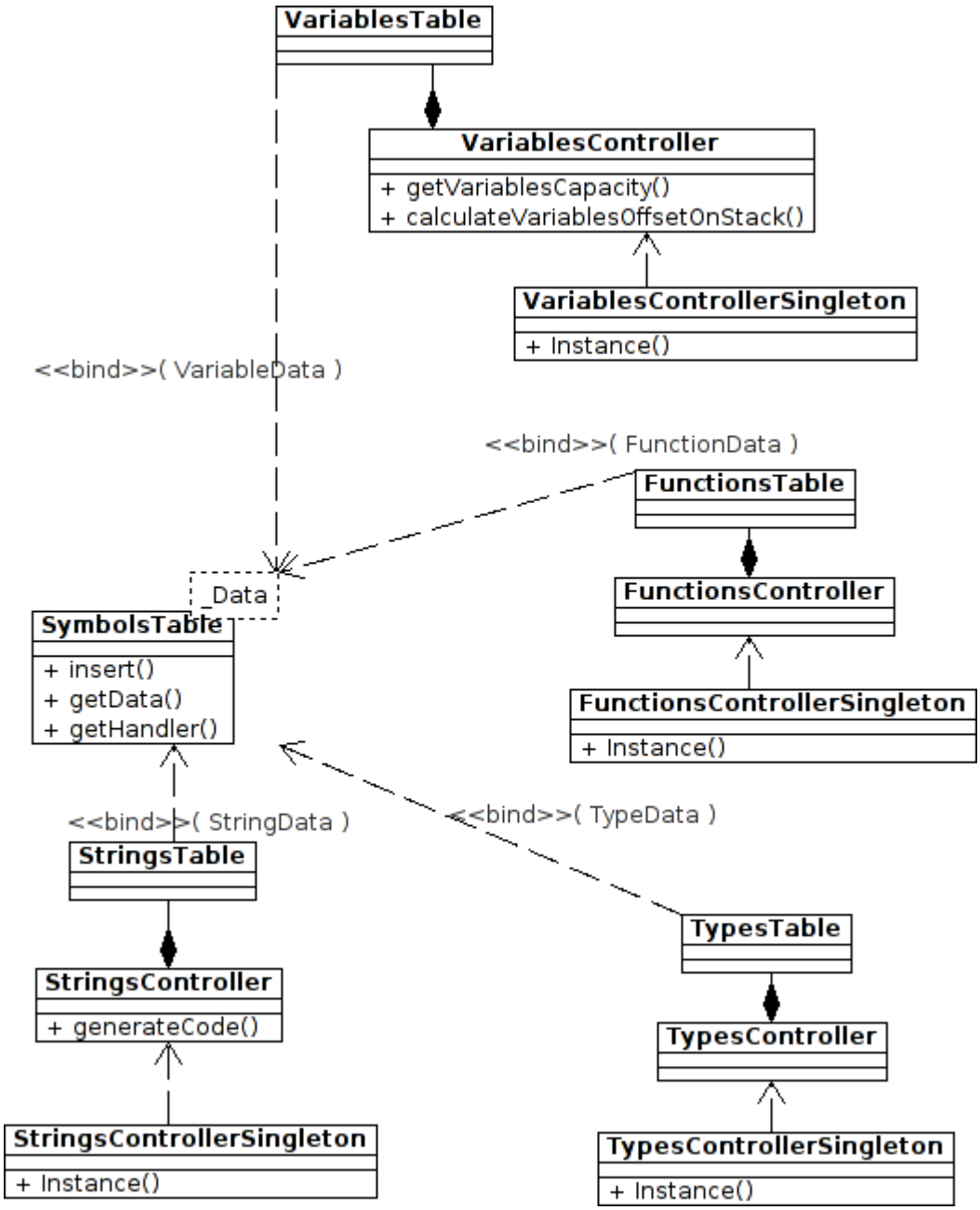


Rys. 8.5 Klasy reprezentujące operatory

Analiza semantyczna

Celem analizy semantycznej jest sprawdzenie drzewa składniowego pod kątem błędów logicznych w programie oraz zebranie informacji niezbędnych do generowania kodu asemblera.

Informacje o funkcjach, typach, literałach napisowych oraz zmiennych gromadzone są w czterech kontrolerach, `VariablesController`, `FunctionsController`, `StringsController` oraz `TypesController`. Wspólna funkcjonalność dla kontrolerów została zebrana w klasie bazowej `SymbolsTable`. Dodatkowo, każdy kontroler dostarcza unikalne metody do obsługi funkcji, literałów napisowych, typów i zmiennych. Wszystkie kontrolery mogą zostać utworzone tylko raz w całej aplikacji. W celu zapewnienia tego wymogu, zastosowano wzorzec projektowy *Singleton*. Dostarczono również wygodny interfejs do tworzenia kontrolerów w postaci klasy fabryki – `SymbolsTableFactory`.



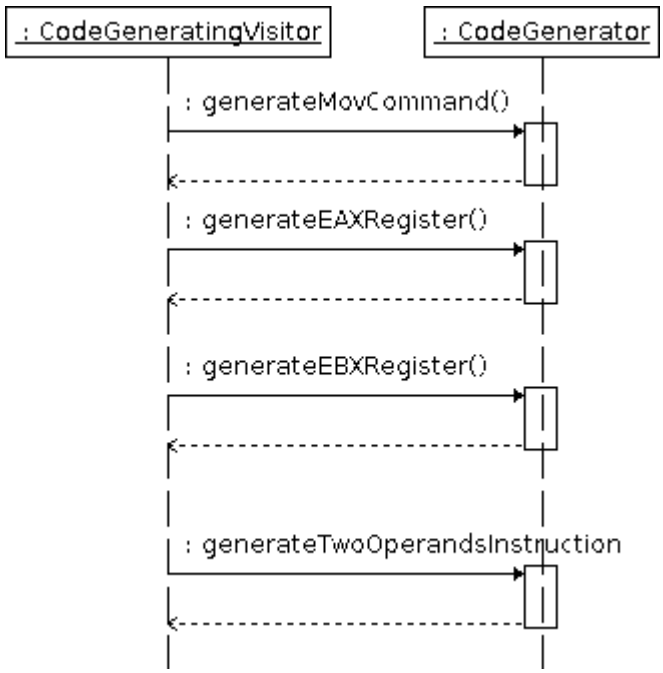
Rys. 8.6 Klasy przechowujące informacje o analizowanym programie

Generowanie kodu asemblera

Generowanie kodu asemblera jest ostatnim etapem kompilacji. Wymaga ono przejścia przez drzewo składniowe, oraz korzystając z informacji zebranych przez poprzednie etapy kompilacji stworzyć kod asemblera.

Generowanie kodu realizowane jest przez klasę `CodeGeneratorVisitor`, która jest przykładem użycia wzorca projektowego *Wizytatora*. Dzięki zastosowaniu tego wzorca, kod generujący asembler zebrany jest w kodzie źródłowym w jednym miejscu oraz jest odseparowany od innych funkcjonalności kompilatora.

Literały języka asembler tworzone są w klasie `CodeGenerator`. `CodeGeneratingVisitor` używa interfejsu klasy `CodeGenerator`, będąc jednocześnie oddzielona od szczegółów składni asemblera. Diagram sekwencji generowania kodu asemblera przedstawiono na Rys. 8.7.



Rys. 8.7 Diagram sekwencji generowania kodu asemblera

Kompilator `mtc` generuje kod asemblera w notacji AT&T. Jest to notacja zgodna ze stosowaną w systemie Linux.

8.7 Wynik działania programu

Do sprawdzenie poprawności działania kompilatora `mtc` przyjęto program, którego kod przedstawiono w Przykład 8.4. Program korzystając z rekurencyjnej oraz iteracyjnej definicji silni, oblicza wartość wyrażenia 5! – 3!, a rezultat tego wyrażenia zwraca jako kod wyjścia do systemu operacyjnego. W wyniku działania skryptu `mtc.sh` uzyskano wydruk przedstawiony na Rys. 8.8. Program zwrócił wartość **114**, czyli poprawny rezultat działania 5! – 3!.

```
Compile...
Compilation finished with... [0] warning(s), [0] error(s).
Generating object file...
Linking...
Program's output...
Program exit with value... 114
Compilation successful.
```

Rys. 8.8 Wydruk działania skryptu `mtc.sh`

Kod asemblera wygenerowany przez kompilator przedstawiony jest na Rys. 8.9. W celu poprawienia czytelności kodu usunięto z niego komentarze.

```
.align 2
.section .text
.globl _start
.globl silnia
      .type silnia, @function
silnia:
      pushl %ebp
      movl %esp, %ebp
      subl $0, %esp
      movl 8(%ebp), %eax
      pushl %eax
      movl $0, %eax
      popl %ebx
      cmpl %eax, %ebx
      sete %al
      cmpl $1, %eax
      je .LIFTRUE1
.LIFFALSE1:
      movl 8(%ebp), %eax
      pushl %eax
      movl 8(%ebp), %eax
      pushl %eax
      movl $1, %eax
      popl %ebx
      xchgl %ebx, %eax
      subl %ebx, %eax
      pushl %eax
      call silnia
      addl $4, %esp
      popl %ebx
      imull %ebx, %eax
      jmp .LIFEND1
.LIFTRUE1:
      movl $1, %eax
.LIFEND1:
      movl %ebp, %esp
      popl %ebp
      ret
.size silnia, .-silnia
.globl main
      .type main, @function
main:
_start:
      pushl %ebp
      movl %esp, %ebp
      subl $0, %esp
      movl $5, %eax
      pushl %eax
      call silnia
      addl $4, %esp
      pushl %eax
      movl $3, %eax
      pushl %eax
      call silnia
      addl $4, %esp
      popl %ebx
      xchgl %ebx, %eax
      subl %ebx, %eax
      movl %ebp, %esp
      popl %ebp
      movl %eax, %ebx
      movl $1, %eax
      int $0x80
.size main, .-main
.align 2

.ident "icc-1.0"
```

Rys. 8.9 Kod asemblera wygenerowany przez `mtc` dla programu z Przykład 8.7

Kod z Rys. 8.9 zawiera dwie funkcje, `silnia` oraz `main`. Każda z funkcji zdefiniowana jest jako symbol publiczny widoczny poza jednostką translacji, poprzez dyrektywę `.globl`. Funkcja `main` jest dodatkowo oznaczona etykietą `_start` wskazującą początek programu.

Funkcja `silnia` oblicza wartość silni w sposób rekurencyjny. Silnia dla argumentu zero zdefiniowana jest jako *silnia*(0) = 1, natomiast dla liczb różnych od zera jako *silnia*(*n*) = *n* * *silnia*(*n* – 1). Kod asemblera oblicza wartość silni wprost z jej rekurencyjnej definicji. Linie wyróżnione kolorem czarnym kopiują wartość argumentu funkcji do rejestru `ebx`.

```
movl 8(%ebp), %eax
pushl %eax
movl $0, %eax
popl %ebx
```

Następnie do rejestru `eax` kopiowana jest wartość 0

```
movl $0, %eax
```

i wartość parametru pamiętana w rejestrze `ebx` porównywana jest z zerem w rejestrze `eax`.

```
cmpl %eax, %ebx
```

Jeżeli wartości są sobie równe, do młodszej części rejestru `eax` wpisywana jest wartość 1

```
sete %al
```

Jeżeli wynik poprzedniego porównania był prawdziwy, wykonuje się instrukcja skoku warunkowego

```
je .LIFTRUE1
```

do miejsca, gdzie funkcja zwraca wartość 1

```
.LIFTRUE1:
      movl $1, %eax
.LIFEND1:
      movl %ebp, %esp
      popl %ebp
      ret
```

W przeciwnym razie, wykonuje się kod który zmniejsza wartość argumentu o jeden

```
movl 8(%ebp), %eax
pushl %eax
movl $1, %eax
popl %ebx
xchgl %ebx, %eax
subl %ebx, %eax
```

i wywołuje funkcję silnia rekurencyjnie

```
pushl %eax
call silnia
```

Funkcja `main` oblicza wartość wyrażenia 5! – 3! korzystając z wcześniej zdefiniowanej funkcji `silnia`. W tym celu wkłada na stos parametr, wartość 5, i wywołuje funkcję `silnia`,

```
movl $5, %eax
pushl %eax
call silnia
```

a zwrócony wynik zapamiętuje w rejestrze `ebx`. Następnie w ten sam sposób funkcja `main` oblicza wartość wyrażenia 3!

```
movl $3, %eax
pushl %eax
call silnia
```

a wynik jest zapamiętywany w rejestrze `eax`. Wynik całego wyrażenia 5! – 3! obliczany jest przez kod

```
xchgl %ebx, %eax
subl %ebx, %eax
```

i zwracany do systemu operacyjnego

```
movl %eax, %ebx
movl $1, %eax
int $0x80
```

9 Wnioski

Podstawowym celem pracy było napisanie aplikacji która wczytuje język wewnętrzny, a następnie generuje kod asemblera dla architektury Intel x86 w notacji AT&T.

Implementacja kompilatora spełnia wszystkie postawione wymagania funkcjonalne. Język wewnętrzny, chociaż udostępnia tylko podstawowe mechanizmy, pozwala na pisanie w nim prostych programów do obliczeń na liczbach całkowitych. Wygenerowany kod asemblera, zawiera komentarze poszczególnych instrukcji, dzięki czemu analiza takiego kodu może pomóc w zrozumieniu działania języka maszynowego.

W fazie analizy przypadków użycia, kompilatorowi nie stawiano wymagań wydajnościowych. Po zaimplementowaniu aplikacji, w wyniku testów wydajnościowych nie stwierdzono jednak występowania wąskich gardeł aplikacji, a czas translacji kodu użytkownika na kod asemblera mieści się przyjętych normach.

Pośród wymagań funkcjonalnych, nie znalazło się również zadania generowania wydajnego kodu asemblera. Algorytm generowania kodu maszynowego dla wyrażeń matematycznych (Rys. 7.8) jest algorytmem prostym, jednak mało wydajnym. Obliczenie wyrażenia zawierającego *n* operatorów, powoduje *2n* odwołań do pamięci (*n* komend `pushl`, *n* komend `popl`), które są dużo wolniejsze niż dostęp do rejestrów procesora.

Kolejną wadą zastosowanego rozwiązania jest przydział wolnych rejestrów. Do obliczenia dowolnego wyrażenia matematycznego wystarczą tylko dwa rejestry *R1* oraz *R2*. Wszystkie wyniki pośrednie przechowywane są na stosie. Nowoczesne procesory posiadają więcej niż dwa rejestry, które nigdy nie będą wykorzystane przez kod wygenerowany kompilatorem `mtc`.

O ile przydział rejestrów jest zadaniem trudnym, o tyle ograniczenie ilości odwołań do stosu można w prosty sposób ograniczyć. Wystarczy obliczać wyrażenie matematyczne od liści drzewa składniowego w stronę korzenia. Obliczane w danym momencie są tylko wartości tych węzłów, dla których znane są wartości wszystkich jego potomków. W ten sposób nie przechowuje się fragmentów wyrażeń matematycznych, których nie można jeszcze obliczyć, na stosie.

Istotnym brakiem w języku wewnętrznym jest brak typu tablicowego. Wygodnym sposobem implementacji typu tablicowego, jest utworzenie typu wskaźnikowego jako bazy. Typ wskaźnikowy znajduje bezpośrednie wsparcie w językach maszynowych, a na jego podstawie można zbudować bardziej wygodny typ tablicowy z operatorem indeksowania. O ile w językach wysokiego poziomu dostęp do elementów tablicy realizowany jest przez operator indeksowania, w języku maszynowym dostęp ten uzyskuje się odpowiednim trybem adresowania (*based indexed addressing*). Zmienna która reprezentuje tablicę, zawiera adres jej pierwszego elementu. Dostęp do *n*-tego elementu uzyskuje się przez przesunięcie w pamięci o *n * rozmiar elementu* bajtów. Rozmiar tablicy nie jest pamiętany przez sam typ. W celu określenia końca tablicy, używa się albo ustalonego znaku za ostatnim elementem, lub ciężar pamiętania rozmiaru tablicy zrzucony jest na programistę.

Tryb tablicowy umożliwia w prosty sposób implementację typu napisowego jako tablicy liter. Napis w języku *C* jest sekwencją znaków typu *char* zakończonych znakiem pustym NULL.

Warto również zauważyć, że poprawnie zdefiniowany typ tablicowy powinien dać się opisać w sposób rekurencyjny. Tablica *n*-wymiarowa jest jak najbardziej poprawną konstrukcją językową. Szczególnym przypadkiem jest *n=0*, kiedy można przyjąć, że typ tablicowy jest pojedynczym elementem. Tablica jednowymiarowa o rozmiarze jeden i pojedynczy element, chociaż zajmują dokładnie tyle samo pamięci są różnymi typami.

10 Literatura

[1] A. A. Aaby, ‘Compiler construction using Flex and Bison’, 2004

[2] A. V. Aho, R. Sethi, J. D. Ullman, *Kompilatory, Reguły*, *metody i narzędzia*, WNT, 2002

[3] A. V. Aho, J. D. Ullman, *Wykłady z informatyki z przykładami w języku C*, Helion, 1995

[4] J. Bartlett, ‘Programming from the Ground Up’, 2003

[5] F. J. F. Benders, J. W. Haaring, T. H. Janssen, D. Meffert, A. C. van OostenRijk, Compiler construction a practicat approach, 2003

[6] K. Dattatri, Język C++ Efektywne programowanie obiektowe, Helion Gliwice, 2005

[7] B. Drozdowski, *Język asembler dla każdego*

[8] J. E. Hopcroft, J. D. Ullman, *Wprowadzenie do teorii automatów, języków i obliczeń*, PWN, 2003

[9] S. Kruk, Asembler w koprocesorze, Mikom 2003

[10] S. Majewski, Wykłady do przedmiotu Języki formalne, AGH Kraków

[11] S. Meyers, Język C++ bardziej efektywny, 1998, WNT Warszawa

[12] H. Sutter, Wyjątkowy język C++ 47 łamigłówek, zadań programistycznych i rozwiązań, 2002, WNT Warszawa

[13] H. Sutter, Wyjątkowy język C++ 40 nowych łamigłówek, zadań programistycznych i rozwiązań, 2005, Helion Gliwice

[14] T. Niemann, A compact guide to lex & yacc

[15] J. Sokołowski, Wykłady do przedmiotu Teoria kompilacji, IPIPAN Gdańsk

[16] A. S. Tanenbaum, *Strukturalna organizacja systemów komputerowych*, wyd. 5, 2006, Helion Gliwice

[17] P. D. Terry, Compilers and compiler generators, 1996

[18] W. M. Waite, G. Goos, Compiler construction

[19] D. Vandevoorde, N. M. Josuttis, C++ Szablony Vademecum profesjonalisty, 2003, Helion Gliwice

[20] Bison, GNU Parser Generator, http://www.gnu.org/software/bison

[21] Boost C++ libraries, http://boost.org

[22] Dokumentacja firmy SUN, http://docs.sun.com

[23] Flex, the fast lexical analyzer, http://flex.sourceforge.net

[24] Strona firmy Intel, http://www.intel.com

[25] Loki library, http://loki-lib.sourceforge.net

[26] Strona Zakładu Informatyki i Teorii Automatów Skończonych Politechniki Śląskiej http://www.zmitac.iinf.polsl.gliwice.pl/

[27] Wolna encyklopedia Wikipedia, http://pl.wikipedia.org

[28] Strona Wydziału Matematyki i Informatyki Uniwersytetu Mikołaja Kopernika w Toruniu http://www-users.mat.uni.torun.pl/

[29] Strona Zakładu Systemów Komputerowych Uniwersytetu Śląskiego http://zsk.tech.us.edu.pl