Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Włodzimierz Bielecki, Piotr Błaszyński

Projektowanie kompilatorów

Teoria i praktyka

Skrypt do wykładów i zajęć laboratoryjnych

Szczecin 2018

Recenzent
EVGENY OCHIN

Opracowanie redakcyjne
ALICJA BERNER

Wydano za zgodą
REKTORA ZACHODNIOPOMORSKIEGO UNIWERSYTETU TECHNOLOGICZNEGO W SZCZECINIE

Autorzy będą bardzo wdzięczni za wszystkie uwagi krytyczne do materiału zawartego w tej książce, które prosimy wysyłać pod jeden z następujących adresów: wbielecki, pblaszynski@wi.zut.edu.pl

ISBN 978-83-7663-263-6

Wydawnictwo Uczelniane Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie
al. Piastów 48, 70-311 Szczecin, tel. 91 449 47 60, e-mail: wydawnictwo@zut.edu.pl
Druk PPH „Zapol" Sobczyk Sp.j., al. Piastów 42, 71-062 Szczecin

Spis treści

Wstęp

Książka ta ma za zadanie przedstawienie podstawowych technik projektowania kompilatorów. **Kompilator** jest to program służący do automatycznego tłumaczenia kodu napisanego w jednym języku (języku źródłowym) na równoważny kod w innym języku (języku wynikowym). Proces ten nazywany jest kompilacją. W książce tej jako forma wyjściowa jest używany kod dla procesorów MIPS, natomiast większa część książki dotyczy ogólnych zasad budowania kompilatorów niezależnych od platformy wyjściowej. Monografia jest adresowana do studentów, doktorantów oraz wszystkich tych, którzy interesują się kompilatorami.

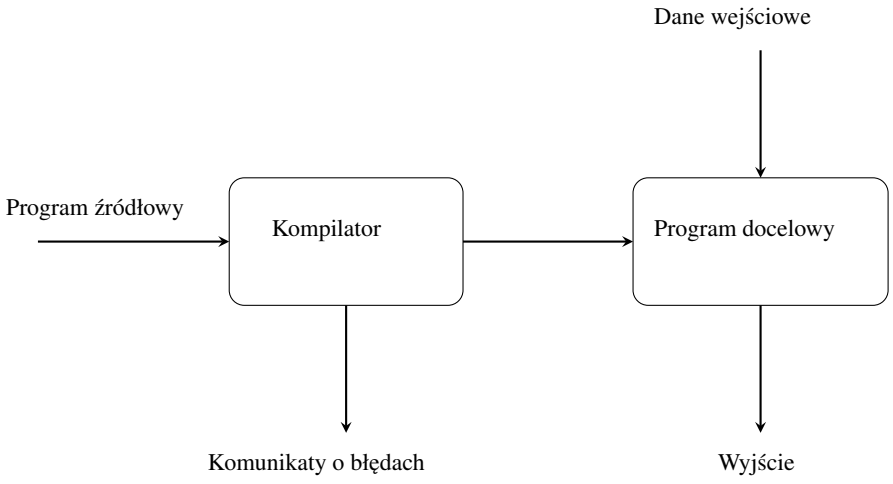
Książka zawiera 15 rozdziałów teoretycznych i 10 rozdziałów zawierających zadania do wykonania. W rozdziale pierwszym przedstawione zostały ogólne założenia i budowa kompilatora. Przedstawiony został również schemat generowania kodu docelowego. Rozdział drugi prezentuje podstawowe pojęcia z zakresu kompilatorów, ze szczególnym uwzględnieniem gramatyki. Rozdział trzeci opisuje proces translacji sterowanej składnią. Rozdział czwarty dotyczy podstawowywch pojęć związanych z analizą leksykalną, natomiast w rozdziale piątym jest przedstawiony schemat rozpoznawania przez analizator leksykalny elementów kodu. Rozdział szósty dotyczy używanych w analizie leksykalnej wyrażeń regularnych i diagramów stanów opisujących automaty. W rozdziale siódmym jest zaprezentowany proces parsowania w analizatorze składniowym. Kontynuacją tego rozdziału jest rozdział ósmy, w którym zaprezentowano metody budowania tablic przydatnych przy parsowaniu. Rozdział dziewiąty zawiera opis klasyfikacji gramatyk oraz zasady analizy wstępującej. W rozdziale dziesiątym temat analizy wstępującej jest rozszerzony o przedstawienie parsera korzystającego z tej metody analizy. W kolejnych rozdziałach są przedstawione elementy budowy kompilatora za pomocą dostępnych narzędzi. W rozdziale jedenastym zaprezentowana jest zasada działania i sposób użycia narzędzia YACC. W rozdziale dwunastym przedstawione są podstawowe zasady działania analizatora semantycznego. W rozdziale trzynastym zaprezentowany jest kod źródłowy prostego translatora wraz z omówieniem. W rozdziale czternastym przestawione są podstawowe konstrukcje używane przy generowaniu kodu MIPS, zaprezentowana jest również organizacja pamięci i zasady korzystania z niej. Rozdział piętnasty przedstawia zasady generowania kodu dla deklaracji zmiennych, wyrażeń, tablic, instrukcji warunkowych i pętli.

W części drugiej zaprezentowane są zadania do samodzielnej pracy. Wykonanie tych zadań prowadzi do implementacji działającego kompilatora autorskiego języka; wyjściem takiego kompilatora jest kod maszynowy dla procesora z rodziny MIPS. Rozdział pierwszy tej części przedstawia przygotowanie do budowy analizatora leksykalnego i ogólne założenia. Rozdział drugi uczy, jak należy zwracać kody leksemów po ich rozpoznaniu przez analizę leksykalną. Rozdział trzeci przedstawia zasady dotyczące definiowania gramatyki i przekazywania wartości semantycznych do analizy składniowej. Rozdział czwarty pokazuje, jak zapisywać w swoim kompilatorze najprostszą reprezentację kodu pośredniego. W rozdziale piątym pokazane jest, jak generować kod dla pojedynczej trójki. W rozdziale szóstym pokazane jest generowanie kodu dla prostych instrukcji wejścia/wyjścia oraz instrukcji warunkowych. W rozdziale siódmym zaprezentowane są instukcje pętli i tablice jednowymiarowe, natomiast w rozdziale ósmym – liczby zmiennoprzecinkowe i tablice wielowymiarowe. Całość dopełniają rozdziały dziewąty i dziesiąty opisujące generowanie kodu dla prostych funkcji, dynamicznych tablic i struktur.

Część I Wykłady

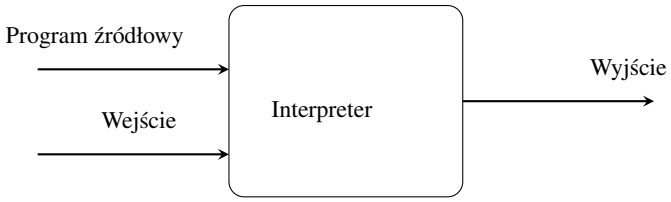
1. Architektura kompilatora

Kompilator jest to program, który odczytuje program w języku źródłowym i tłumaczy go na program równoważny w języku docelowym. Interpreter, zamiast produkowania programu docelowego, bezpośrednio wykonuje czynności określone w programie źródłowym. Program docelowy, produkowany przez kompilator, jest zwykle znacznie szybszy niż proces produkowania wyniku przez interpreter. Rysunek 1.1 pokazuje schemat działania kompilatora.



Rys. 1.1. Schemat działania kompilatora

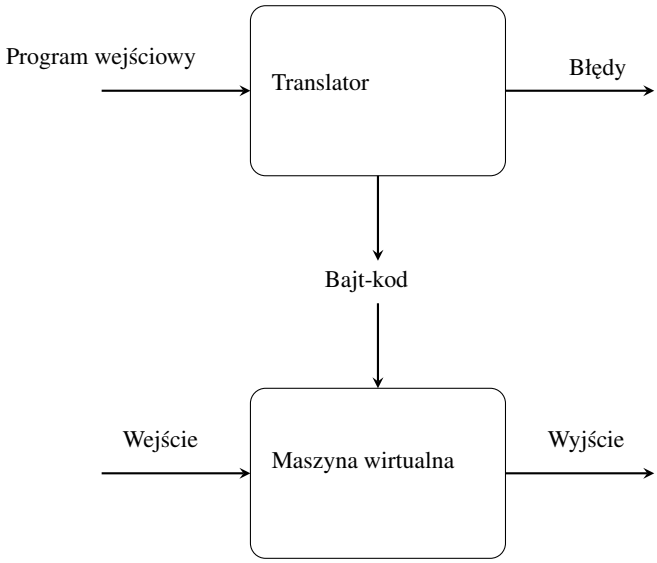
Interpreter jednak zazwyczaj daje lepszą diagnostykę błędów niż kompilator, ponieważ wykonuje instrukcję programu źródłowego instrukcja po instrukcji. Schemat działania interpretera pokazuje rysunek 1.2.



Rys. 1.2. Schemat działania interpretera

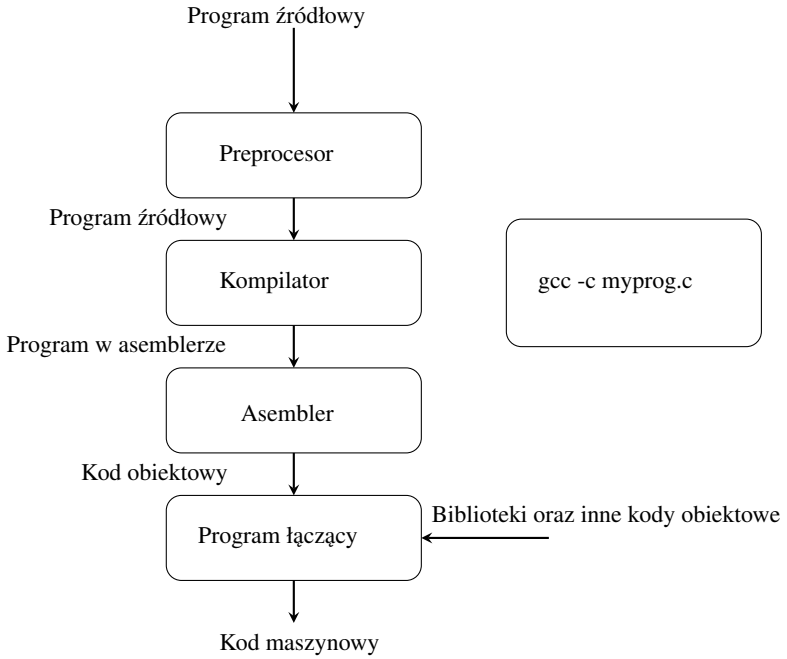
Ważną rolą kompilatora jest zgłaszanie wszelkich błędów w programie źródłowym, które są wykrywane podczas procesu tłumaczenia.

Wirtualne procesory języka Java łączą kompilację i interpretację. Program źródłowy w języku Java jest najpierw kompilowany do postaci pośredniej zwanej bajtkodami (*bytecodes*). Bajtkody następnie są interpretowane przez maszynę wirtualną. Na rysunku 1.3 pokazany jest schemat działania kompilatora hybrydowego.

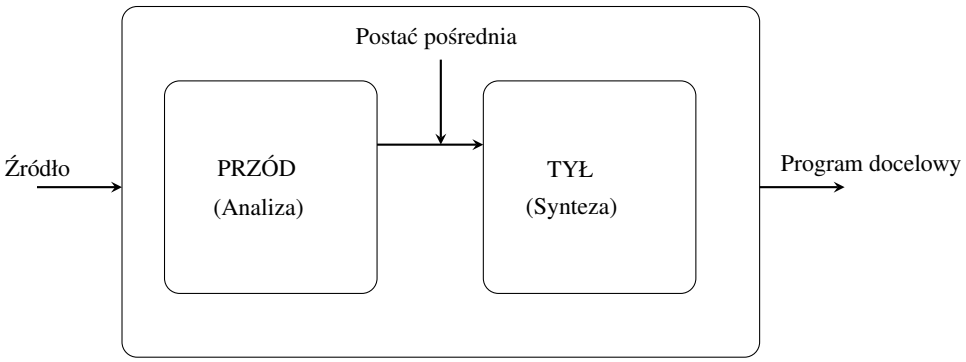


Rys. 1.3. Schemat działania kompilatora hybrydowego

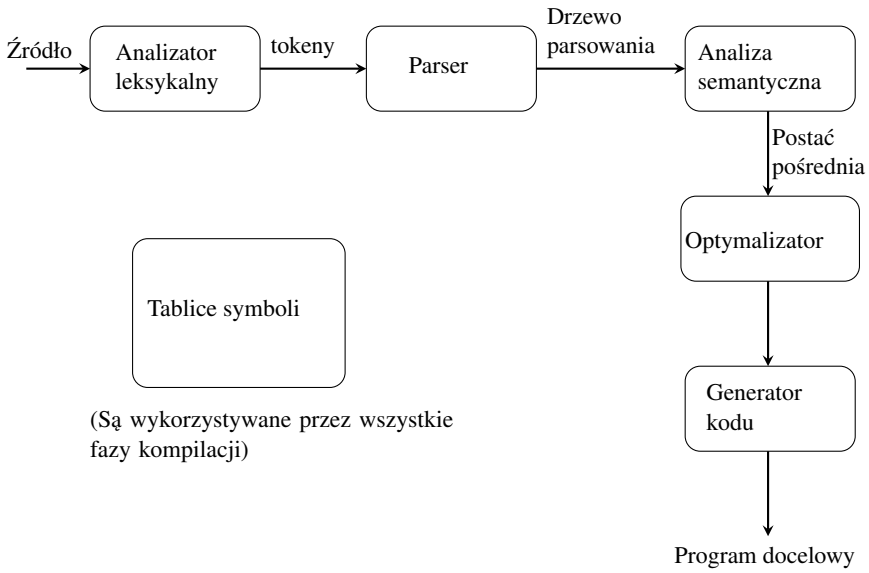
Oprócz kompilatora może być potrzebnych kilka innych programów, aby utworzyć wykonywalny program docelowy. Kolejność przetwarzania programu żródłowego w kod maszynowy pokazuje rysunek 1.4. Ogólna architektura kompilatora (podział na przód i tył) pokazana jest na rysunku 1.5. Natomiast szczegóły budowy kompilatora, z podziałem na poszczególne elementy, pokazano na rysunku 1.6.



Rys. 1.4. Schemat działania kompilatora – kolejność przetwarzania kodu źródłowego



Rys. 1.5. Architektura kompilatora

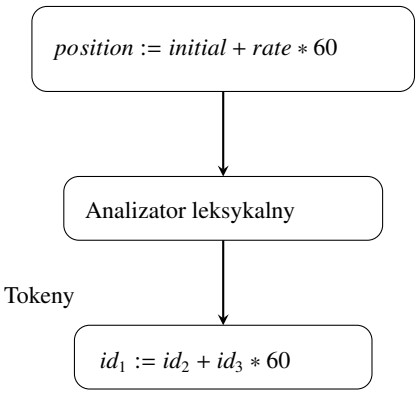


Rys. 1.6. Architektura kompilatora – szczegóły

Analiza leksykalna. Analizator leksykalny czyta znaki z programu źródłowego i grupuje je w sekwencje, które reprezentują leksemy. Dla każdego leksemu analizator leksykalny produkuje token o postaci: $< token - name, attribute - value >$. Na przykład załóżmy, że program źródłowy zawiera instrukcję przypisania: $position := initial + rate * 60$. Analizator leksykalny produkuje następujący wynik:

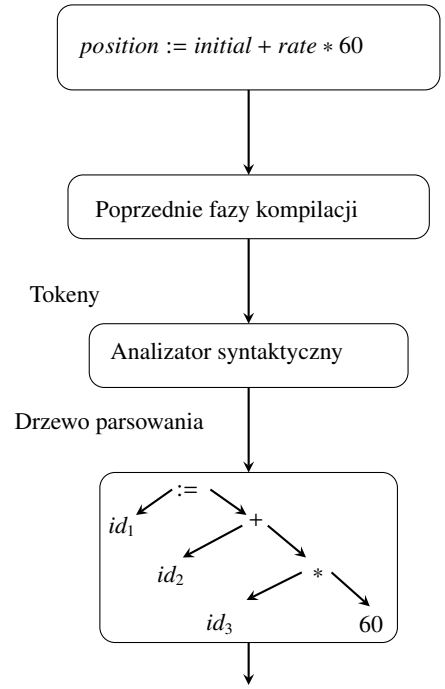
1. *position* jest leksemem, dla którego jest tworzony token: $< id, 1 >$, gdzie *id* jest to symbol abstrakcyjny oznaczający identyfikator; 1 jest adresem, pod którym tablica symboli przechowuje leksem *position* oraz dodatkowe atrybuty, na przykład typ danych.
2. Symbol przypisania := jest leksemem, dla którego jest produkowany token <:=>. Ponieważ ten token nie wymaga atrybutu, drugi składnik jest pominięty.
3. *initial* jest leksemem, dla którego jest tworzony token $< id, 2 >$; 2 jest adresem, pod którym tablica symboli przechowuje leksem *initial*.
4. + jest leksemem, dla którego jest produkowany token $< + >$.
5. *rate* jest leksemem, dla którego jest tworzony token $< id, 3 >$; 3 jest adresem, pod którym tablica symboli przechowuje leksem *rate*.
6. $*$ jest leksemem odwzorowywanym na token $< * >$.
7. 60 jest leksemem odwzorowywanym na token $< 60 >$.

Na rysunku 1.7 przedstawiony jest wynik produkowany przez analizator leksykalny dla instrukcji: $position := initial + rate * 60$



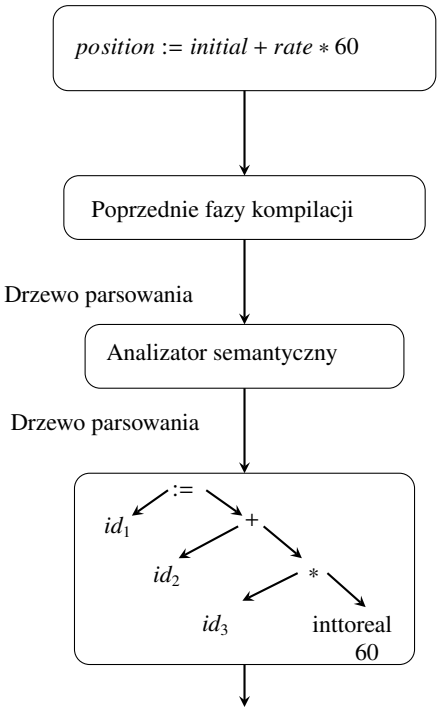
Rys. 1.7. Schemat działania analizatora leksykalnego

Analiza syntaktyczna. Parser wykorzystuje pierwsze składniki tokenów, produkowane przez analizator leksykalny, aby utworzyć reprezentację pośrednią, która przedstawia strukturę gramatyczną strumienia tokenów. Typową reprezentacją składni jest drzewo syntaktyczne, w którym każdy węzeł wewnętrzny reprezentuje operację, natomiast „dzieci" tego węzła stanowią argumenty operacji. Na podstawie utworzonego drzewa parser decyduje, czy składnia programu jest poprawna. Na wyjściu parsera uzyskano wynik w postaci drzewa parsowania pokazany na rysunku 1.8.



Rys. 1.8. Schemat działania analizatora syntaktycznego

Analiza semantyczna. Analizator semantyczny korzysta z drzewa parsowania, wykorzystuje informacje przechowywane w tablicy symboli i sprawdza program źródłowy pod względem spójności semantycznej, zdefiniowanej przez język programowania. Ponadto gromadzi informacje o typach zmiennych i zapisuje je w drzewie parsowania lub w tablicy symboli do wykorzystania podczas kolejnych etapów generacji kodu pośredniego. Ważną częścią analizy semantycznej jest kontrola typów – kompilator sprawdza, czy każdy operator ma dopasowane argumenty. Na przykład wiele języków programowania wymaga, żeby indeksy tablicy były liczbami całkowitymi; kompilator musi zgłosić błąd, jeśli do reprezentacji indeksu tablicy jest używana liczba zmiennoprzecinkowa. Specyfikacja języka może pozwalać na konwersję typów, znaną jako wymuszenie (*coercion*). Na przykład operator arytmetyczny może być zastosowany do pary liczb całkowitych lub pary liczb zmiennoprzecinkowych. Jeśli operandy nie należą do tego samego typu danych, to jeden z nich może być konwertowany do typu drugiego operandu. Na rysunku 1.9, pokazującym schemat działania analizatora semantycznego, operator *inttoreal* konwertuje liczbę całkowitą na liczbę zmiennoprzecinkową.

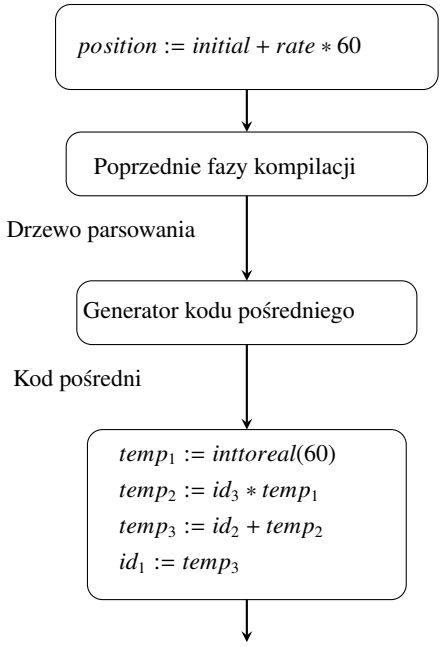


Rys. 1.9. Schemat działania analizatora semantycznego

Generacja kodu pośredniego. W procesie tłumaczenia programu źródłowego na kod docelowy kompilator może utworzyć jedną reprezentację lub kilka reprezentacji pośrednich, które mogą mieć różne formy. Na przykład drzewa składniowe są popularną formą reprezentacji pośredniej. Schemat działania generatora kodu pośredniego pokazany jest na rysunku 1.10.

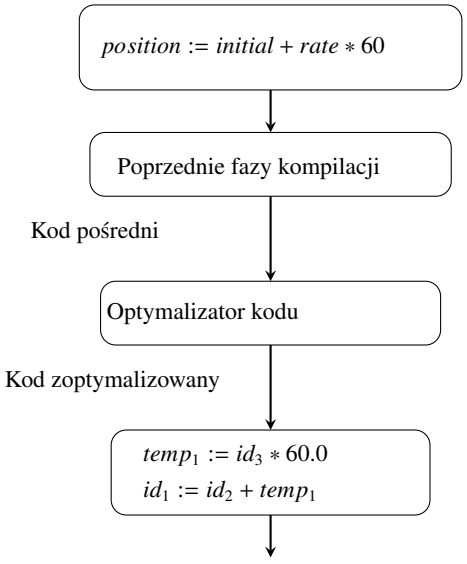
Drugą popularną formą jest kod trójadresowy:

```
1   t1 = inttoreal(60)
2   t2 = id3 * t1
3   t3 = id2 + t2
4   id1 = t3
```



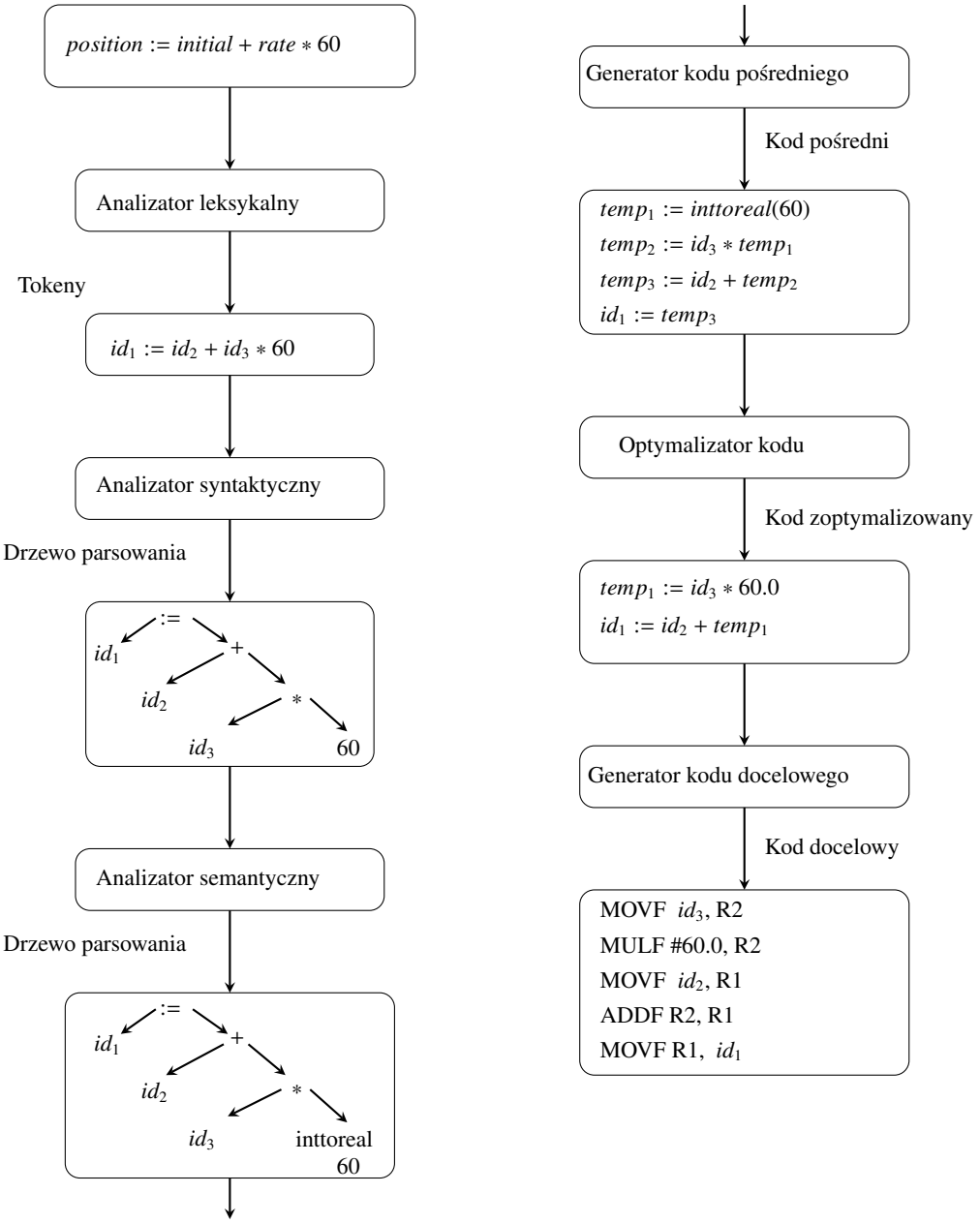
Rys. 1.10. Generacja kodu pośredniego

Optymalizacja kodu. Polega ona na redukcji liczby instrukcji i/lub zmniejszeniu zapotrzebowania na pamięć (zmniejszenie liczby zmiennych tymczasowych). Optymalizacja kodu może być fazą opcjonalną; schemat działania tej fazy kompilatora pokazany jest na rysunku 1.11.



Rys. 1.11. Schemat z optymalizacją kodu

Generacja kodu. Generator kodu tłumaczy reprezentację pośrednią programu na program w języku docelowym. Jeśli programem docelowym ma być kod maszynowy, to generator kodu musi przydzielić pamięć (rejestry, pamięć operacyjną) dla każdej zmiennej zdeklarowanej w programie źródłowym. Następnie każda instrukcja postaci pośredniej jest tłumaczona na sekwencję instrukcji maszynowych. Aspektem kluczowym generowania kodu maszynowego jest optymalny przydział rejestrów do przechowywania zmiennych. Na rysunku 1.12 pokazany jest pełny schemat budowy kompilatora; zawiera on wszystkie fazy kompilacji.



Rys. 1.12. Generowanie kodu docelowego

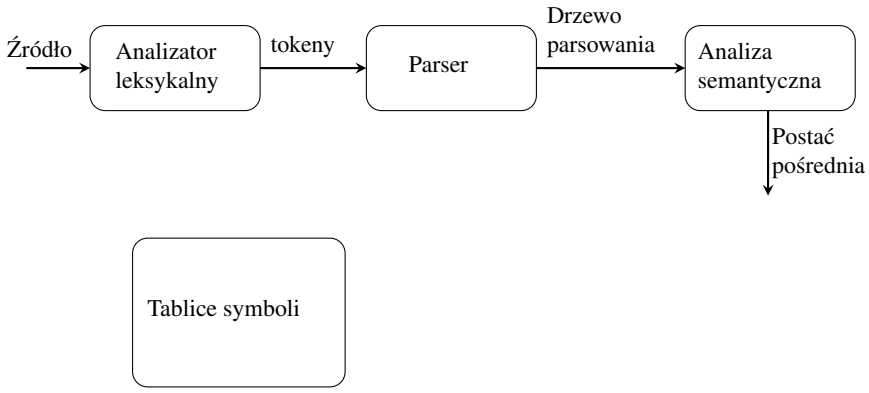
Tablica symboli. Jest ona strukturą danych zawierającą rekord dla każdej nazwy zmiennej, wraz z polami do przechowywania atrybutów tej zmiennej. Tablica symboli powinna być zaprojektowana tak, aby kompilator mógł szybko znaleźć rekordy dla każdej nazwy oraz szybko zapisać lub pobrać dane z tego rekordu.

Grupowanie faz kompilacji. Przedstawione fazy kompilatora pokazują jego logiczną organizację. W implementacji kompilatora fazy te mogą być grupowane w jedną większą fazę. Na przykład analiza leksykalna, analiza syntaktyczna i analiza semantyczna mogą być połączone w jedną fazę; w tej fazie czynności wszystkich analiz są wykonywane jednocześnie pod kontrolą analizatora syntaktycznego (kompilacja sterowana składnią).

2. Pojęcie gramatyki, drzewo parsowania, łączność operatorów, gramatyka jednoznaczna

Składnia języka programowania. Opisuje ona właściwą strukturę programu, natomiast semantyka języka określa co program robi – czyli jaki jest jego sens.

Struktura kompilatora. Każdy kompilator ma przód i tył; struktura przodu kompilatora jest pokazana na rysunku 2.1.



Rys. 2.1. Struktura przodu kompilatora

Języki programowania

$$1 + 2 * 3 = 7 \tag{2.1}$$

$$1 + *23 = ??? \tag{2.2}$$

Czy ciągi 2.1 i 2.2 są poprawnie zbudowanym wyrażeniem arytmetycznym? Do odpowiedzi na to pytanie potrzebna jest gramatyka, czyli zbiór produkcji.

Pojęcie produkcji. Produkcja jest to para uporządkowana, na przykład: $S \rightarrow 1$.

Gramatyka. Jest to zbiór produkcji:

$$S \rightarrow AB \tag{2.3}$$

$$A \rightarrow 1 \tag{2.4}$$

$$A \rightarrow A1 \tag{2.5}$$

$$B \rightarrow 0 \tag{2.6}$$

$$B \rightarrow B0 \tag{2.7}$$

S (wzór 2.3) jest symbolem początkowym. Symbole nieterminalne $N = S, A, B$ występują po prawej stronie produkcji; mogą wystąpić również po lewej stronie produkcji (ale nie muszą). Symbole terminalne $T = 0, 1$ występują tylko po prawej stronie produkcji.

Wyprowadzenia:
- na podstawie 2.3 : $S \rightarrow AB$
- na podstawie 2.4 : $AB \rightarrow 1B$
- na podstawie 2.6 : $1B \rightarrow 10$

czyli: $S \rightarrow 10$, więc z S można wyprowadzić 10, stosując jedną produkcję lub większą liczbę produkcji.

Specyfikacja BNF. Backus-Naur Form (BNF) jest formą używaną do wyrażenia gramatyk bezkontekstowych. Nazwa wywodzi sie od nazwisk naukowców zajmujących się opisem gramatyk języków programowania; byli to John Warner Backus i Peter Naur. Specyfikacja BNF jest to zbiór produkcji o postaci: *symbol* → *expression*, gdzie *symbol* jest to nieterminal, natomiast *expression* jest to sekwencja jednego symbolu lub większej liczby symboli terminalnych i/lub nieterminalnych. Większą liczbę sekwencji oddzielamy kreską pionową '|', wskazując wybór. Symbole, które nigdy nie pojawiają się po lewej stronie produkcji, są to **terminale** (są one pogrubione). Symbole pojawiające się po lewej stronie produkcji są to **nieterminale** (są one wyróżnione kursywą). Przykład produkcji: *stmt* → *if*(*expr*) *stmt else stmt*.

Gramatyka bezkontekstowa (*A context-free grammar*) zawiera:
1. Zbiór symboli terminalnych (terminali). Terminale są to elementarne symbole języka zdefiniowanego przez gramatykę.
2. Zbiór symboli nieterminalnych (zmienne syntaktyczne, nieterminale). Każdy nieterminal reprezentuje sekwencję terminali w sposób, który poznamy w dalszej części.
3. Zbiór produkcji, gdzie każda produkcja składa się z nieterminala, zwanego głową lub lewą stroną produkcji, ze strzałki oraz z sekwencji terminali i/lub nieterminali, nazywanej ciałem lub prawą stroną produkcji.
4. Jeden wyznaczony symbol nieterminalny zwany symbolem startowym.

Gramatyka jest to skończony zbiór produkcji, w którym lewa strona pierwszej produkcji wskazuje symbol startowy. Przykład gramatyki:

```
list -> list + digit                          (1)
list -> list - digit                          (2)
list -> digit                                 (3)
digit -> 0 | 1| 2| 3| 4| 5| 6| 7| 8| 9        (4)
```

Ciała trzech produkcji, których lewa strona jest tym samym symbolem list, można pogrupować:

```
   list -> list + digit | list - digit | digit
```

Według przedstawionej definicji symbole terminalne są następujące: $+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ Nieterminalami są *list* i *digit*. Symbolem startowym jest *list*. Przykład gramatyki – zapis formalny :

```
G = < {list,digit}, {+,-,0,1,2,3,4,5,6,7,8,9}, P, list >
Z produkcjami P =
 list -> list + digit
 list -> list - digit
 list -> digit
 digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  8 | 9
```

Ciąg symboli (napis) jest to sekwencja składająca się z zera lub większej liczby symboli. Ciąg, który nie zawiera żadnego symbolu, jest nazywany napisem pustym ε.

Wyprowadzenia. Korzystając z gramatyki, wyprowadzamy napisy, zaczynając zawsze od symbolu startowego, i wielokrotnie zastępujemy pojedynczy nieterminal prawą stroną produkcji, której lewa strona jest zastępowanym nieterminalem. Dla gramatyki:

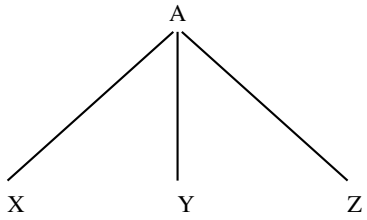
```
list -> list + digit | list - digit | digit
digit -> 0 |1| 2| 3| 4| 5| 6| 7|  8| 9
```

możemy wyprowadzić:

```
list -> digit -> 0
list -> list + digit -> digit + digit -> 1 + digit -> 1+2
```

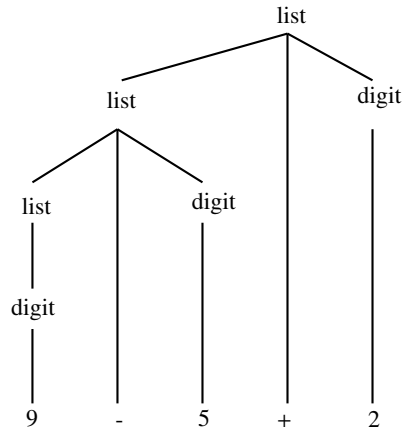
Parsowanie ma na wejściu ciąg terminali i „zastanawia się”, jak wyprowadzić ten ciąg z symbolu startowego gramatyki; jeśli nie można wyprowadzić takiego ciągu, to jest raportowany błąd składni. Drzewo parsowania pokazuje obrazowo, w jaki sposób z symbolu startowego gramatyki można wyprowadzić zdanie wejściowe.

Dla produkcji: $A \rightarrow XYZ$ drzewo parsowania ma postać jak na rysunku 2.2:



Rys. 2.2. Drzewo parsowania

Dla napisu 9 – 5 + 2 i gramatyki G drzewo parsowania ma postać jak na rysunku 2.3:



Rys. 2.3. Drzewo parsowania

Formalnie w przypadku gramatyki bezkontekstowej drzewo parsowania ma następujące właściwości:

1. Korzeń jest oznaczony symbolem startowym.
2. Każdy liść jest oznaczony przez terminal lub ε.
3. Każdy węzeł wewnętrzny jest oznaczony przez nieterminal.
4. Jeśli A jest nieterminalem, który oznacza pewien węzeł wewnętrzny, a $X1, X2, ..., Xn$ są etykietami dzieci tego węzła od lewej do prawej strony, to istnieje produkcja $A- > X1X2..Xn$, gdzie każde $X1, X2, ..., Xn$ oznacza symbol terminalny lub nieterminalny. W szczególnym przypadku, jeśli $A- > \varepsilon$ jest produkcją, to węzeł oznaczony jako A ma jedno dziecko ε.

Terminologia związana z drzewem. Drzewo składa się z jednego lub większej liczby węzłów. Węzły mogą mieć etykiety, które zazwyczaj są symbolami gramatycznymi. Gdy rysujemy drzewo, często reprezentujemy węzły tylko przez te etykiety. Dokładnie jeden węzeł jest korzeniem. Wszystkie węzły, oprócz korzenia, mają unikatowego rodzica; korzeń nie ma rodzica. Gdy rysujemy drzewo, to umiejscawiamy rodzica zawsze powyżej dziecka; rodzic i dziecko są połączone krawędzią.

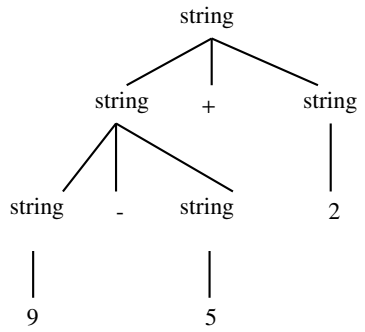
- Jeśli węzeł N jest rodzicem węzła M, to M jest dzieckiem N.
- Dzieci jednego węzła nazywane są rodzeństwem.
- Węzeł bez dzieci jest to liść.
- Inne węzły – te z jednym dzieckiem lub z większą liczbą dzieci – są to węzły wewnętrzne.
- Potomkiem węzła N jest sam węzeł N lub dziecko N, lub dziecko dziecka itd.
- Mówimy, że węzeł N jest przodkiem węzła M, jeśli M jest potomkiem N.

Niejednoznaczność. Dla danej gramatyki, dla ciągu terminali może istnieć więcej niż jedno drzewo parsowania. Taka gramatyka jest nazywana gramatyką niejednoznaczną. Ponieważ napis, dla którego istnieje więcej niż jedno

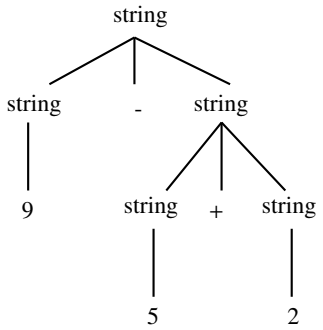
drzewo parsowania zwykle ma więcej niż jedno znaczenie, musimy zaprojektować gramatykę jednoznaczną lub używać gramatyk niejednoznacznych z dodatkowymi zasadami rozwiązywania niejednoznaczności. Korzystając z gramatyki:

```
string -> string + string | string - string| 0|1|2|3|4|5|6|7|8|9,
```

dla napisu 9 – 5 + 2 można utworzyć dwa drzewa parsowania (rysunki 2.4 i 2.5):



Rys. 2.4. Pierwsze drzewo parsowania



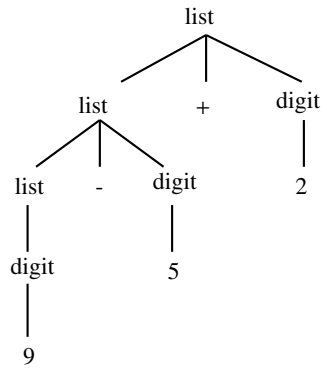
Rys. 2.5. Drugie drzewo parsowania

Łączność operatorów. Zgodnie z konwencją 9 + 5 + 2 jest równoważne z (9 + 5) + 2, natomiast 9 – 5 – 2 jest równoważne z (9 – 5) – 2. Gdy argument 5 ma operatory po jego lewej i prawej stronie, reguły są potrzebne do podjęcia decyzji, który z operatorów odnosi się do tego argumentu. Mówimy, że operator + jest łączny lewostronnie, ponieważ argument, który ma znak plus po obu jego stronach, należy do operatora po jego lewej stronie. W większości języków programowania cztery operatory arytmetyczne: dodawanie, odejmowanie, mnożenie i dzielenie są łączne lewostronnie. Niektóre operatory, takie jak potęgowanie, są łączne prawostronnie. Operator przypisania = też jest łączny prawostronnie, co oznacza, że wyrażenie $a = b = c$ traktuje się w taki sam sposób jak wyrażenie $a = (b = c)$. Ciągi takie, jak $a = b = c$, są generowane przez następującą gramatykę:

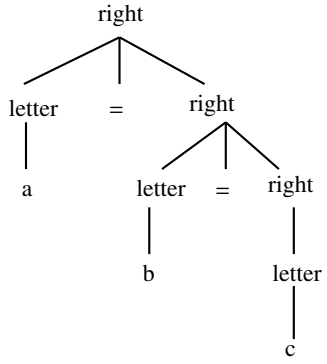
```
right -> letter = right | letter
letter -> a | b | ... | z
```

Według tych reguł zapis $a = b = c$ jest równoważny z zapisem $a = (b = c)$. Struktura drzew dla łączności lewostronnej i prawostronnej jest różna; w pierwszym przypadku drzewo rośnie – w dół i w lewo; w drugim przypadku – w dół i w prawo. Kontrast między drzewem parsowania dla operatora – łącznego lewostronnie i drzewem parsowania dla operatora = łącznego prawostronnie jest pokazany na rysunkach 2.6 i 2.7.

```
list -> list + digit || list - digit | digit
digit -> 0 | 1 | 2 | ... | 9
```



Rys. 2.6. Łączność operatorów lewostronna



Rys. 2.7. Łączność operatorów prawostronna

Według tych reguł zapis $9 - 5 + 2$ jest równoważny z zapisem $(9 - 5) + 2$.

Pierwszeństwa operatorów. Rozważmy wyrażenie $9 + 5 * 2$. Istnieją dwie możliwe interpretacje tego wyrażenia: $(9 + 5) * 2$ lub $9 + (5 * 2)$. Zasadę łączności stosuje się do wystąpień tego samego operatora, a więc nie rozwiązuje ona dwuznaczności. Gramatykę dla wyrażeń arytmetycznych można skonstruować na podstawie tabeli reprezentującej pierwszeństwa operatorów. Zaczynamy od czterech operatorów arytmetycznych i tabeli pierwszeństwa, pokazującej operatory w kolejności rosnącego priorytetu. Operatory na tej samej linii mają taką samą łączność i pierwszeństwo:

– łączne lewostronnie: + −
– łączne lewostronnie : ∗ /

Gramatyka jednoznaczna. Zasada tworzenia gramatyki jednoznacznej: należy dodatkowo wprowadzić $N + 1$ nieterminali, gdzie N jest to liczba poziomów pierwszeństwa. W naszym przykładzie $N = 2$. Tworzymy dwa nieterminale *expr* i *term* dla dwóch poziomów pierwszeństwa oraz dodatkowy nieterminal $factor$ do generowania podstawowych jednostek w wyrażeniach. Podstawowe jednostki w wyrażeniach są to cyfry i wyrażenia w nawiasach: $factor- > digit|(expr)$. Rozważmy teraz operatory binarne ∗ i /, które mają najwyższy priorytet. Odpowiednie produkcje mają postać:

```
term
    -> term * factor
     | term / factor
     | factor
```

Produkcje, które odpowiadają za wyrażenia z operatorami + i −, mają postać:

```
expr
   -> expr + term
    | expr - term
    | term
```

Poprzez połączenie powyższych gramatyk uzyskujemy następującą gramatykę jednoznaczną:

```
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digit | ( expr )
```

Kroki użyte do przetwarzania wyrażenia 2 + 3 ∗ 5:

Krok 1:

```
factor -> digit | ( expr )
```

Krok 2:

```
term -> term * factor
    | term / factor
    | factor
```

Krok 3:

```
expr -> expr + term
    | expr - term
    | termactor
    | factor
```

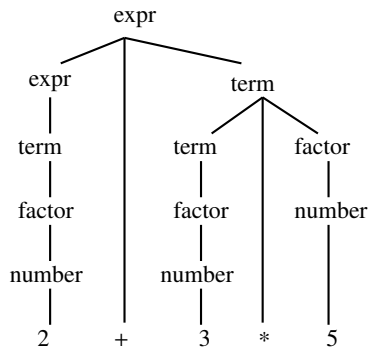
Krok 4:

```
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digit | ( expr )
```

Użyte produkcje:

```
expr -> expr + term | term
term -> term * factor | factor
factor -> number | ( expr )
```

Zdanie 2+3*5 jest parsowane w sposób jednoznaczny – tak jak to przedstawiono na rysunku 2.8:



Rys. 2.8. Gramatyka jednoznaczna

3. Translacja sterowana składnią

Translacja sterowana składnią. Odbywa się poprzez dołączenie zasad (reguł) lub fragmentów kodu do produkcji w gramatyce. Na przykład w przypadku produkcji: $expr \rightarrow expr_1 + term$ możemy przetwarzać $expr$ wykorzystując strukturę produkcji zgodnie z poniższym pseudokodem:

```
1 dopasuj expr1;
2     przetwarzaj expr1;
3 dopasuj term;
4     przetwarzaj term;
5 dopasuj +;
6     przetwarzaj +;
```

Atrybut jest to pewna wartość powiązana z konstrukcją języka programowania. Przykładami atrybutów są typy danych wyrażeń, liczba instrukcji wygenerowanego kodu lub lokalizacja pierwszej instrukcji generowanego kodu.

Schemat translacji sterowany składnią jest to gramatyka, w przypadku której do każdej produkcji jest przypisany fragment kodu – akcja semantyczna. Fragmenty kodu są wykonywane, natomiast produkcja jest wykorzystywana przez parser.

Notacja postfiksowa może być zdefiniowana w następujący sposób:

1. Jeżeli E oznacza zmienną lub stałą, to notacją postfiksową dla E jest samo E.
2. Jeżeli E jest wyrażeniem postaci $E1\ op\ E2$, gdzie op jest operatorem binarnym, to notacją postfiksową dla E jest $E1'\ E2'\ op$, gdzie $E1'$ i $E2'$ są notacjami postfiksowymi odpowiednio dla $E1$ i $E2$.
3. Jeżeli E jest wyrażeniem o postaci $(E1)$, to notacja postfiksowa dla E jest taka sama jak notacja postfiksowa dla $E1$.

Przykład: Notacją postfiksową dla wyrażenia $(9 - 5) + 2$ jest $9\ 5\ -\ 2+$. Oznacza to, że przekład dla 9, 5 i 2 jest reprezentowany przez te same stałe, zgodnie z regułą (1). Tłumaczeniem $9 - 5$ zgodnie z regułą (2) jest $9\ 5-$. Tłumaczeniem dla $(9-5)$ zgodnie z regułą (3) jest $9-5$. Nawiasy nie są potrzebne w notacji postfiksowej, ponieważ pozycja i liczba argumentów operatorów w tej notacji w sposób jednoznaczny określają kolejność wykonywania operatorów.

Atrybuty syntezowane. Atrybuty kojarzymy z nieterminalami i terminalami. Żeby obliczyć atrybuty nieterminali, dodajemy reguły do produkcji. Reguły te opisują, w jaki sposób obliczane są atrybuty w węzłach drzewa parsowania.

Definicje przekładu sterowanego składnią formułuje się poprzez:

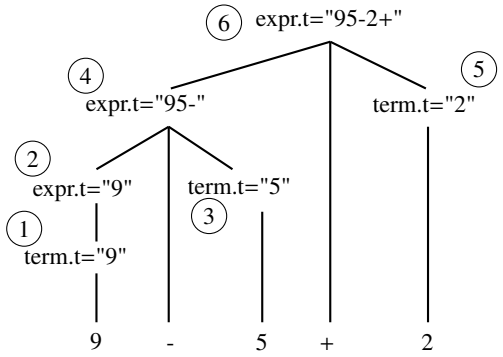
1. Zbiór atrybutów dla każdego symbolu gramatycznego.
2. Zbiór reguł semantycznych do obliczania wartości atrybutów związanych z symbolami występującymi w produkcji.

Atrybuty mogą być obliczone w następujący sposób: Dla danego ciągu wejściowego x tworzymy drzewo parsowania. Następnie stosujemy reguły semantyczne w każdym węźle drzewa parsowania w następujący sposób: Załóżmy, że węzeł N w drzewie parsowania jest oznaczony symbolem X. Wtedy zapisujemy wartość atrybutu a w tym węźle jako $X.a$. Drzewo parsowania, pokazujące wartości atrybutów w każdym węźle, nazywamy drzewem parsowania z przypisami. Wartość atrybutu syntezowanego dla węzła N w drzewie parsowania oblicza się na podstawie atrybutów jego dzieci i atrybutu własnego.

Przykład obliczania atrybutów: Poniżej podano definicję przekładu sterowanego składnią do tłumaczenia postaci infiksowej na postfiksową:

```
1  expr -> expr1 + term     expr.t := expr1.t || term.t ||"+"
2  expr -> expr1- term      expr.t := expr1.t ||term.t || "-"
3  expr -> term             expr.t := term.t
4  term -> 0                term.t := "0"
5  term -> 1                term.t := "1"
6  term -> 2                term.t := "2"
7  term -> 3                term.t := "3"
8  ...                      ...
9  ...                      ...
10 term -> 9                term.t := "9"
```

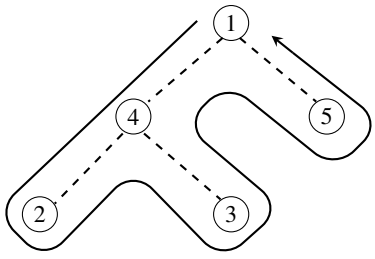
Wartości atrybutów w węzłach drzewa parsowania dla ciągu: $9 - 5 + 2$ podano na rysunku 3.1. Liczby w okręgach oznaczają kolejność dopasowania poszczególnych atrybutów:



Rys. 3.1. Atrybuty syntezowane

Przechodzenie przez drzewo jest stosowane do obliczania atrybutów oraz wykonywania fragmentów kodu (akcji) w schemacie translacji. Przechodzenie przez drzewo zaczyna się od jego korzenia, następnie odwiedza się każdy węzeł drzewa w pewnej kolejności.

Przechodzenie drzewa w głąb (ang. *depth-first traversal*) zaczyna się od korzenia; polega na odwiedzaniu dzieci każdego węzła w dowolnej kolejności, niekoniecznie od lewej do prawej. Ogólny schemat tego przechodzenia zaprezentowano na rysunku 3.2. Nazywa się ono „w głąb", ponieważ odwiedza nieodwiedzone dziecko węzła, gdy tylko może, a więc odwiedza węzły występujące jak najdalej od korzenia i tak szybko, jak jest to możliwe. Na rysunku 3.3 przedstawiono przykład przechodzenia drzewa w głąb; liczby w okręgach określają kolejność przechodzenia.

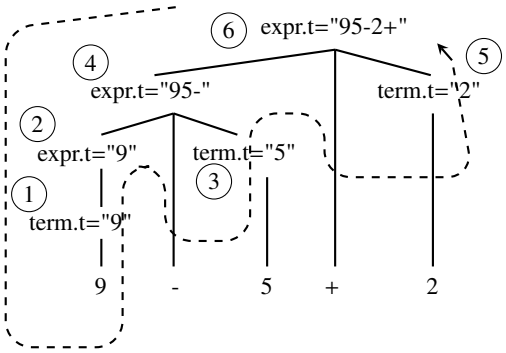


Rys. 3.2. Przechodzenie drzewa w głąb – schemat

Definicja sterowana składnią nie narzuca żadnej konkretnej kolejności obliczania atrybutów w drzewie parsowania; każda kolejność, która oblicza atrybut *a* po wszystkich innych atrybutach, od których *a* zależy, jest akceptowalna.

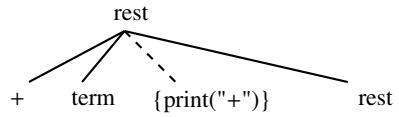
Przechodzenie drzewa w głąb:

```
procedure visit(node N)
 {
   for (each child C of N, from left to right)
         visit (C) ;
     zastosuj reguły semantyczne w węzle N;
 }
```



Rys. 3.3. Przechodzenie drzewa w głąb – przykład

Schemat translacji sterowany składnią jest to notacja dla określenia translacji, która powstaje po dołączeniu fragmentów kodu do produkcji w gramatyce. Fragmenty kodu, dodane do produkcji, nazywamy akcjami semantycznymi. Pozycja, gdzie akcja ma zostać wykonana, jest pokazana przez umieszczenie jej w klamrach po prawej stronie produkcji: $rest \rightarrow + \; term \; print(" + ")rest$. Dodatkowy węzeł reprezentuje akcję semantyczną; jest połączony na rysunku 3.4 linią przerywaną z węzłem odpowiadającym „głowie” (lewej stronie).



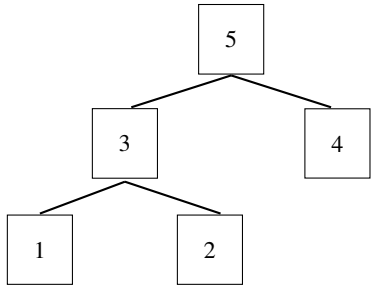
Rys. 3.4. Dodatkowy węzeł w schemacie translacji

Akcje do translacji na notację postfiksową, akcje semantyczne po prawej stronie, w nawiasach klamrowych:

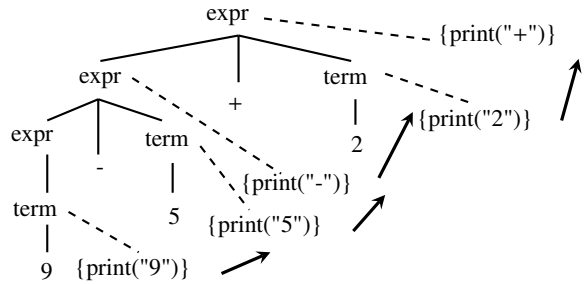
```
expr -> expr1 + term     {print('+')}
expr -> expr1- term      {print('-')}
expr -> term
term -> 0                {print ('0')}
term -> 1                {print ('1')}
...
...
term -> 9                {print ('9')}
```

Implementacja schematu translacji musi zapewnić, że akcje semantyczne zostaną wykonane w kolejności, w jakiej pojawiają się w trakcie przechodzenia drzewa *post-order*. Zanim odwiedzimy dany wierzchołek, odwiedzamy wszystkich jego potomków. Poruszamy się od najniższej generacji w górę. Liczby w wierzchołkach na rysunku 3.5, pokazującym przechodzenie drzewa w głąb, oznaczają kolejność ich odwiedzania.

Implementacja nie musi w rzeczywistości konstruować drzewa parsowania, jeżeli zapewnia, że wszystkie akcje semantyczne są wykonywane tak, jak byśmy konstruowali drzewo syntaktyczne; następnie akcje przedstawione na rysunku 3.6 są wykonywane zgodnie z przechodzeniem *post-order*. Rysunek ten pokazuje translację wyrażenia na jego notację postfiksową.



Rys. 3.5. Przechodzenie drzewa w głąb – przykład



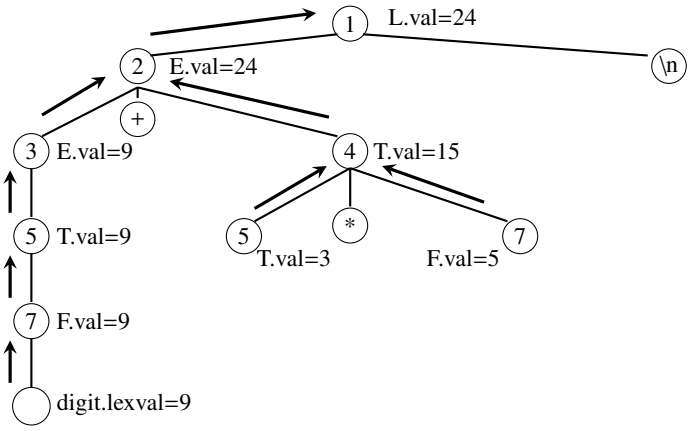
Rys. 3.6. Translacja wyrażenia 9 – 5 + 2 do postaci 9 5 – 2+

Poniższe produkcje i akcje semantyczne definiują prosty kalkulator:

```
1 L ::= E \n       L.val = E.val
2 E ::= E1 + T     E.val = E1.val + T.val
3 E ::= T          E.val = T.val
4 T ::= T1 * F     T.val = T1.val * F.val
5 T ::= F          T.val = F.val
6 F ::= ( E )      F.val = E.val
7 F ::= digit      F.val = digit.lexval
```

W powyższym kodzie produkcje występują po lewej stronie, a akcje semantyczne – po prawej. Na rysunku 3.7 jest przedstawiony sposób obliczenia wyrażenia $9 + 3 * 5\backslash n$. Liczby w okręgach oznaczają numer produkcji (numer linii) z powyższego kodu.

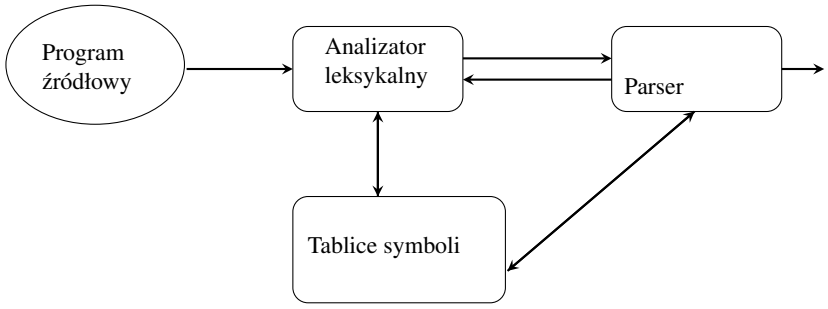
Uwaga: Produkcja numer 6 ($F ::= (E)$) nie została wykorzystana do tworzenia drzewa parsowania.



Rys. 3.7. Obliczanie wyrażenia 9 + 3 * 5\n

4. Analiza leksykalna 1

Analizator leksykalny odczytuje znaki z wejścia, rozpoznaje leksemy i produkuje tokeny. Wraz z symbolem terminalnym, który jest używany przez parser, token zawiera dodatkowe informacje w postaci wartości atrybutów. Token jest to terminal wraz z dodatkową informacją. Rola analizatora leksykalnego w procesie kompilacji jest pokazana graficznie na rysunku 4.1.



Rys. 4.1. Rola analizatora leksykalnego

Sekwencja znaków wejściowych, która zawiera pojedynczy token, nazywa się leksemem. Można zatem powiedzieć, że analizator leksykalny izoluje parser od reprezentacji znakowej symbolu, co oznacza, że parser dostaje tokeny, a nie znaki. Głównym zadaniem analizatora leksykalnego jest wczytywanie znaków z programu źródłowego, rozpoznawanie leksemów i produkowanie tokenów (znaczników) dla każdego leksemu. Inne zadania to eliminowanie komentarzy i „znaków białych": spacja, znak nowej linii, znak tabulacji. Kolejnym zadaniem jest współudział w obsłudze błędów generowanych przez kompilator. Na przykład analizator leksykalny może śledzić liczbę linijek kodu, liczbę znaków każdej linijki i przekazywać te dane do kompilatora.

Dlaczego analizator leksykalny jest tworzony osobno?

1. Prostota projektowania analizatora leksykalnego (w stosunku do projektowania analizatora syntaktycznego) jest najważniejszym czynnikiem.
2. Zwiększona jest wydajność kompilatora. Opracowane są wyspecjalizowane bardzo wydajne techniki analizy leksykalnej. Ponadto techniki buforowania do wczytywania znaków wejściowych mogą znacznie przyspieszyć kompilator.
3. Kompilator ma zwiększoną przenośność, czyli może być stosowany na różnych platformach. Specyficzne dla urządzeń wejścia osobliwości mogą być uwzględnione tylko w analizatorze leksykalnym; pozostałe części kompilatora zostają bez zmian.

Tokeny, wzorce, leksemy. Wzorzec jest opisem postaci, którą leksem może przyjąć. W przypadku słów kluczowych wzorzec jest po prostu ciągiem znaków, które tworzą słowa kluczowe. Leksem jest ciągiem znaków w programie źródłowym, który pasuje do jakiegoś wzorca. Jest to niepodzielny element programu, dlatego jest nazywany również atomem. Przykładowe tokeny są przedstawione w tabeli 4.1.

Tabela 4.1. Tokeny, wzorce, leksemy

Token	Leksem	Wzorzec
if	if	if
id	abc, n, count,...	litera + cyfra
NUMBER	3.14, 1000	stała numeryczna
;	;	;

Dla kodu:

```
printf ("Total = %d\n", score) ;
```

zarówno *printf* i *score* są leksemami pasującymi do wzorca tokenu *id*, natomiast $Total = \%d\backslash n$ jest to leksem pasujący do literału (dosłowny tekst). W wielu językach programowania następujące przypadki obejmują większość tokenów:

1. Jeden token dla każdego słowa kluczowego. Wzorzec dla słowa kluczowego jest taki sam jak słowo kluczowe.
2. Tokeny dla operatorów: indywidualny token dla każdego operatora lub jeden token dla grupy operatorów (przykładowo operatorów relacji).
3. Jeden token reprezentujący wszystkie identyfikatory.
4. Jeden lub większa liczba tokenów reprezentujących stałe, takie jak liczby i literały.
5. Tokeny dla każdego z symboli interpunkcyjnych, takich jak lewy i prawy nawias, przecinek, średnik.

Token posiada opcjonalne atrybuty. Najważniejszym przykładem jest token *id*, z którym musimy skojarzyć dużo informacji: typ danych, wymiar tablicy, liczbę elementów tablicy, miejsce w programie, w którym zmienna pojawia się po raz pierwszy.

Atrybuty tokenów. Atrybuty są przechowywane w tablicy symboli. Jednym z atrybutów jest zatem również wskaźnik do wpisu w tablicy symboli dla identyfikatora.

Przykład: Dla instrukcji przypisania w Fortranie:

```
E = M * C ** 2
```

mamy następujący wynik produkowany przez analizator leksykalny:

```
< id,
  pointer to symbol-table entry for E
>
< assign_op >
< id,
  pointer to symbol-table entry for M
>
< mult_op >
< id,
  pointer to symbol-table entry for C
>
< exp-op >
< number,
  integer value 2
>
```

Błędy leksykalne (*lexical errors*):

– Wykrywanie błędów $fi(a == f(x))$...
– Raportowanie błędów.
– Usuwanie błędów.

Usuwanie błędów. Załóżmy, że analizator leksykalny podczas rozpoznawania leksemu nie może kontynuować swojego działania, ponieważ żaden ze wzorców nie pasuje. Analizator może usunąć kolejne znaki z pozostałego wejścia, aż dopasuje wczytywane znaki do jakiegoś wzorca. Analizator leksykalny, który będziemy tworzyć, pozwala na rozpoznawanie liczb, identyfikatorów i „znaków białych” (spacji, tabulatorów i znaków nowej linii) w wyrażeniach.

Schemat translacji:

```
expr ->expr + term          { print ('+')}
         | expr - term      { print('-') }
         | term
term -> term * factor       { print('*') }
         | term / factor    { print ('/')}
         | factor
factor -> ( expr )
           | num            { print (num. value) }
           | id             { print (id. lexeme) }
```

Większość języków umożliwia dowolną ilość przestrzeni białej między leksemami. Komentarze mogą być traktowane jako przestrzeń biała. Jeśli przestrzeń biała jest eliminowana przez analizator leksykalny, to parser nie będzie musiał brać pod uwagę znaków białych. Alternatywnie można uwzględnić znaki białe w gramatyce, ale to znacznie zwiększa złożoność parsera.

Pseudokod rozpoznawania i usuwania znaków białych:

```
for ( ; ; peek = next input character ) {
   if ( peek is a blank or a tab ) do nothing;
   else if ( peek is a newline ) line = line+1;
   else break;
}
```

Czytanie z wyprzedzeniem. Analizator leksykalny może wymagać odczytywania znaków wejściowych z wyprzedzeniem, zanim zdecyduje, jaki ma być leksem właściwy. Na przykład analizator leksykalny dla C lub Javy po rozpoznaniu znaku > musi odczytać następny znak. Jeśli następnym znakiem jest =, to znak > jest częścią sekwencji znaków >=, reprezentujących leksem (operator) „większe lub równe". Inaczej znak > sam tworzy leksem „większy niż"; w takim przypadku analizator leksykalny odczytał jednak jeden znak za dużo. Ogólne podejście do czytania znaków wejścia z wyprzedzeniem jest oparte na zastosowaniu bufora wejściowego, z którego analizator leksykalny może odczytać znak i zapisać go z powrotem. Bufory wejściowe mogą być uzasadnione również w odniesieniu do efektywności analizatora, ponieważ pobieranie ciągu znaków jest zwykle bardziej wydajne niż odczyt jednego znaku na raz. Wskaźnik śledzi część wejścia, która już została przeanalizowana; przejście do poprzedniego znaku jest realizowane przez przesunięcie wskaźnika do tyłu. Jeden znak odczytany z wyprzedzeniem zazwyczaj wystarcza, więc prostym rozwiązaniem jest użycie zmiennej, powiedzmy o nazwie *peek*, do przechowywania następnego znaku wejściowego. Analizator leksykalny czyta z wyprzedzeniem tylko wtedy, gdy musi. Operatora $'*'$ można użyć, aby nie odczytywać następnego znaku. W takich przypadkach wartością zmiennej *peek* jest znak spacji, który może być pominięty, gdy analizator jest wywoływany, aby znaleźć następny leksem. Za każdym razem, gdy w wyrażeniu pojawia się pojedyncza cyfra, rozsądne wydaje się wczytywanie kolejnych cyfr w celu rozpoznania liczby całkowitej, ponieważ jest ona sekwencją cyfr. Stałe całkowite mogą być reprezentowane poprzez utworzenie symbolu terminalnego, powiedzmy o nazwie *num* dla każdej stałej, lub poprzez wprowadzenie składni stałych całkowitych do gramatyki. Praca łączenia cyfr w liczbę z reguły należy do zadań analizatora leksykalnego, w związku z czym liczby mogą być traktowane jako pojedyncze jednostki (leksemy) w trakcie parsowania i tłumaczenia kodu źródłowego.

Rozpoznawanie stałych. Gdy w strumieniu wejściowym pojawia się ciąg cyfr, analizator leksykalny przekazuje do parsera token, który składa się z terminala *num* wraz z atrybutem – rozpoznanej liczby całkowitej. Dla napisu wejściowego: 31 + 28 + 59 analizator leksykalny produkuje następujący wynik: $<$ *num*, 31 $><$ + $><$ *num*, 28 $><$ + $><$ *num*, 59 $>$. Pseudokod do rozpoznawania stałych zaprezentowano poniżej:

```
if ( peek holds a digit ) {
    v = 0;
    do {   v = v * 10 + integer value of digit peek;
           peek = next input character;
    } while ( peek holds a digit ) ;
    return token (num, v);
}
```

5. Analiza leksykalna 2

Rozpoznawanie identyfikatorów i słów kluczowych. Większość popularnych języków programowania używa słów kluczowych, na przykład: for, do, if, $while$, itd., które są reprezentowane przez ciągi znaków. Ciągi znaków są również wykorzystywane do tworzenia nazw zmiennych, tablic, funkcji itd. Chcąc uprościć parser, gramatyki traktują identyfikatory jako terminal, powiedzmy *id*, za każdym razem, gdy identyfikator pojawi się na wejściu.

Na przykład dla wejścia: *count* = *count* + *increment*; parser dostanie od analizatora leksykalnego następujący ciąg tokenów: *id* = *id* + *id*. Dla tokena *id* atrybutem jest leksem reprezentujący identyfikator.
Dla wejścia:

count = *count* + *increment*;

analizator leksykalny produkuje:

< *id*,"*count*" >
<=>
< *id*,"*count*" >
< + >
< *id*,"*increment*" ><; >

Słowa kluczowe generalnie spełniają zasady tworzenia identyfikatorów, a zatem jest potrzebny mechanizm do podjęcia decyzji: leksem reprezentuje słowo kluczowe czy identyfikator. Problem jest łatwiejszy do rozwiązania, jeśli słowa kluczowe są zastrzeżone – nie mogą one być wykorzystywane jako identyfikatory. Ciąg znaków tworzy identyfikator, jeżeli nie reprezentuje słowa kluczowego. Analizator leksykalny rozwiązuje ten problem za pomocą tablicy do przechowywania ciągów znaków, czyli symboli.

1. Tablica symboli może izolować resztę kompilatora od reprezentacji ciągów, fazy kompilatora mogą korzystać z referencji lub wskaźnika do łańcucha w tabeli. Referencje / wskaźniki mogą być bardziej efektywne niż manipulowanie samymi ciągami.
2. Tablica symboli może być inicjalizowana słowami zarezerwowanymi.

Gdy analizator leksykalny rozpoznaje leksem, który może stanowić identyfikator, najpierw sprawdza, czy leksem jest przechowywany w tablicy symboli. Jeżeli jest przechowywany, to zwraca token przechowywany w tablicy; w przeciwnym razie tworzy i zwraca token, którego pierwszym elementem jest *id*. Tablica symboli może być zaimplementowana jako tablica haszująca z użyciem klasy o nazwie *Hashtable*. Na przykład: *Hashtable words = new Hashtable*();. W poniższym fragmencie programu został przedstawiony podstawowy schemat działania analizatora leksykalnego i gromadzenia symboli w tablicy symboli:

```
1  if ( peek holds a letter ) {
2      collect letters or digits into a buffer b;
3      s = string formed from the characters in b;
4      w = token returned by words.get(s);
5      if ( w is not null )    return w;
6      else {
7          Enter the key-value pair  (s, <id, s>) into words
8          return token <id, s>;
9      }
10 }
```

Poniżej przedstawiony jest pseudokod funkcji *scan*, która zwraca tokeny:

```
Token scan() {
    skip white space;
    handle numbers;
    handle reserved words and identifiers;
    /* if we get here, treat read-ahead character
        peek as a token */
    Token t = new Token(peek);
    peek = blank
    return t;
```

Analizator leksykalny w C:

```
int lineno = 1;
int tokenval = NONE;

int lexan ()
{
    int t;
    while (1) {
        t = getchar ();
        if (t == ' ' || t == '\t');
        else if (t == '\n')
            lineno++;
        else if (isdigit (t))
        {
            tokenval = t - '0';
            t = getchar();
            while (isdigit(t))
            {
                tokenval = tokenval*10 + t - '0';
                t = getchar();
            }
            ungetc (t, stdin);
            return NUM;
        }
        else
        {
            tokenval = NONE;
            return t;
        }
    }
}
```

Tablica symboli jest wykorzystywana do przechowywania zmiennych oraz słów kluczowych. Jest inicjalizowana poprzez wstawienie do niej słów kluczowych:

```
int insert(const char* s, int t); /* zwraca indeks w tablicy symboli dla
    nowego leksemu s i tokenu t */
int lookup(const char* s); /* zwraca indeks wpisu dla leksemu s lub 0
   gdy nie znaleziono */
```

```
insert("div", DIV);
insert("mod", MOD);
```

Analizator leksykalny z tablicą symboli:

```
int lexan () {
    int t;
    while (1) {
        t = getchar ();
        if (t == ' ' || t == '\t');
        else if (t == '\n')
            lineno++;
        else if (isdigit (t)) {
            ungetc (t, stdin);
            scanf ("%d", &tokenval);
            return NUM;
        } else if (isalpha (t)) {
            int p, b = 0;
            while (isalnum (t)) {
                lexbuf[b] = t;
                t = getchar ();
                b++;
                if (b >= BSIZE) error ("compiler error");
            }
            lexbuf[b] = EOS;
            if (t != EOF) ungetc (t, stdin);
            p = lookup (lexbuf);
            if (p == 0) p = insert (lexbuf, ID);
            tokenval = p;
            return symtable[p].token;
        } else if (t == EOF) return DONE;
        else {
            tokenval = NONE;
            return t;
        }
    }
}
```

Ciągi znaków i języki (*Strings and languages*). Alfabet jest to dowolny skończony zbiór symboli. Typowymi przykładami symboli są litery, cyfry i znaki interpunkcyjne. Zbiór $\{0,1\}$ jest alfabetem binarnym. Znaki tablicy kodów ASCII tworzą ważny przykład alfabetu. Napis (ciąg znaków, łańcuch) nad pewnym alfabetem jest to skończony ciąg symboli z tego alfabetu. Długość napisu s, zapisywana jako $|s|$, jest to liczba wystąpień dowolnych symboli w napisie s. Na przykład napis *banan* ma długość 5. Pusty ciąg znaków, oznaczany jako ε, jest ciągiem o zerowej długości.

Bardzo szeroka definicja języka: Język jest to zbiór napisów nad pewnym ustalonym alfabetem.

Przykład: Dla alfabetu $L = A, ..., Z$, zbiór $A, B, C, BF, \ldots, ABZ, \ldots$ jest językiem zdefiniowanym przez alfabet L. Należy zauważyć, że definicja „języka" nie wymaga, aby każdy napis miał jakiś sens.

Pojęcia związane z napisami

Prefiks (przedrostek) napisu s jest to dowolny napis uzyskany przez usunięcie zera lub większej liczby symboli z końca s. Na przykład *ban*, *banana* i ε są prefiksami napisu *banana*.

Sufiks (przyrostek) napisu s jest to dowolny napis uzyskany przez usunięcie zera lub większej liczby symboli z początku s. Na przykład *nana*, *banana* i ε są sufiksami napisu *banana*.

Podnapis (podciąg spójny) napisu s jest uzyskiwany poprzez usuwanie jakichkolwiek przedrostków i przyrostków z s. Na przykład *banana*, *nan* i ε są podnapisami napisu *banana*.

Złączenie (*concatenation*) napisów x i y jest to napis xy utworzony przez dodanie y na końcu do x. Na przykład jeśli $x = dog$ i $y = house$, to $xy = doghouse$. Dla napisu pustego są prawidłowe równości: $\varepsilon s = s\varepsilon = s$.

Podnoszenie napisu do potęgi definiujemy w następujący sposób: $s^0 = \varepsilon \;\; s^i = s^{i-1}s$ dla $i > 0$.

Przykłady: $s^1 = s$, $s^2 = ss$, $s^3 = sss$.

Operacje na językach:

Suma(*union*):

$L \cup M = s|s \in L \text{ or } s \in M$

Złączenie (*concatenation*):

$LM = xy|x \in L \text{ and } y \in M$

Potęgowanie (*exponentiation*):

$L^0 = \varepsilon; L^i = L^{i-1}L$

Domknięcie Kleene'a (*Kleene closure*):

$L^* = \cup_{i=0,...,\infty} L^i$

Domknięcie dodatnie (*positive closure*):

$L^+ = \cup_{i=1,...,\infty} L^i$

Wyrażenia regularne (*regular expressions*). Podstawowe wyrażenia regularne:

- ε jest wyrażeniem regularnym oznaczającym język {ε}
- $a \in \Sigma$ jest wyrażeniem regularnym oznaczającym język {a}

Jeśli r i s są wyrażeniami regularnymi oznaczającymi języki $L(r)$ i $M(s)$, to:

- $r|s$ jest wyrażeniem regularnym oznaczającym język $L(r) \cup M(s)$,
- rs jest wyrażeniem regularnym oznaczającym język $L(r)M(s)$,
- r^* jest wyrażeniem regularnym oznaczającym język $L(r)^*$,
- (r) jest wyrażeniem regularnym oznaczającym język $L(r)$.

Czyli wyrażenia regularne są budowane z mniejszych wyrażeń regularnych w sposób rekurencyjny. Wyrażenia regularne mogą zawierać zbędne pary nawiasów. Możemy usunąć niektóre pary nawiasów, jeśli przyjmiemy następującą konwencję:

- Operator jednoargumentowy * (operator domknięcia) ma najwyższy priorytet i jest łączny lewostronnie.
- Złączenie ma drugi najwyższy priorytet i jest łączne lewostronnie.

Przykład: $\Sigma = a, b$.

1. Wyrażenie regularne $a|b$ oznacza język $\{a, b\}$.
2. $(a|b)(a|b)$ oznacza aa, ab, ba, bb, czyli język, którego elementami są napisy o długości 2.
3. a^* oznacza język, którego elementy są złożeniem zera lub większej liczby symboli a: $\{\varepsilon, a, aa, aaa, ...\}$.
4. $(a|b)^*$ oznacza język, którego elementy są złożeniem zera lub większej liczby symboli a lub b: $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$.
5. $a|a * b$ oznacza język $\{a, b, ab, aab, aaab, ...\}$

Przykłady:

Rozważmy alfabet $\Sigma = \{a\}$

Jakie jest wyrażenie regularne, które oznacza język, którego każde słowo ma długość nieparzystą?

Rozważmy alfabet $\Sigma = \{a\}$

$a(aa)*$ jest wyrażeniem regularnym, które oznacza język, którego każde słowo ma długość nieparzystą.

Rozważmy alfabet $\Sigma = \{a, b\}$

Jakie jest wyrażenie regularne, które oznacza język, którego każde słowo zaczyna się od litery b?

Rozważmy alfabet $\Sigma = \{a, b\}$

$b(a|b)^*$ jest wyrażeniem regularnym, które oznacza język, którego każde słowo zaczyna się od litery b.

Rozważmy alfabet $\Sigma = \{a, b\}$

Jakie jest wyrażenie regularne, które oznacza język, którego każde słowo musi zaczynać się na literę a, a kończy się na literę b?

Rozważmy alfabet $\Sigma = \{a, b\}$

$b(a|b)^*a$ jest wyrażeniem regularnym, które oznacza język, którego każde słowo musi zaczynać się na literę b, a kończy się na literę a.

Rozważmy alfabet $\Sigma = \{a, b, c\}$

Jakie jest wyrażenie regularne, które oznacza język: $L = \{a, c, ab, cb, abb, cbb, abbb, cbbb, abbbb, cbbbb, ...\}$?

Rozważmy alfabet $\Sigma = \{a, b, c\}$

$((a|c)b^*)$ oznacza język: $L = \{a, c, ab, cb, abb, cbb, abbb, cbbb, abbbb, cbbbb, ...\}$.

6. Analiza leksykalna 3

Definicje regularne (*regular definitions*). Jeśli Σ jest alfabetem symboli podstawowych, to definicją regularną jest sekwencja :

$d_1 -> r_1$

$d_2 -> r_2$

...

$d_n -> r_n$

gdzie:

1. Każde d_j jest to unikatowy symbol (d_j są różnymi symbolami), nienależący do alfabetu.
2. Każde r_i jest wyrażeniem regularnym nad symbolami z alfabetu $\{d_1, d_2, ..., d_i - 1\}$.

Definicje regularne nie mogą być rekurencyjne:

digits -> digit digits | digit

Powyższe to błąd!

Przykład: Definicja regularna dla nazw w języku C:

```
letter -> A | B |...| Z | a | b |...| z |...
digit -> 0| 1 |...| 9
id -> letter_ (letter_ | digit )*,
```

gdzie: *letter_* oznacza dowolną literę lub znak podkreślenia.

Rozszerzenia wyrażeń regularnych. Rozszerzenia, które zostały po raz pierwszy wprowadzone do narzędzia Lex:

1. Jedno wystąpienie lub większa liczba wystąpień. Jednoskładnikowy postfiksowy operator + reprezentuje domknięcie dodatnie wyrażenia regularnego i jego języka. Jeśli r jest wyrażeniem regularnym, to $(r)+$ oznacza język $(L(r))+$. Operator + (domknięcie dodatnie) ma ten sam priorytet i łączność co operator $*$ (domknięcie) oznaczający zero lub większą liczbę znaków. Dwa użyteczne prawa algebraiczne:
   $r* = r + | \varepsilon$
   $r+ = rr* = r * r$
2. Zero wystąpień lub jedno wystąpienie. Jednoskładnikowy postfiksowy operator ? oznacza „zero lub jedno wystąpienie". To oznacza, że $r?$ jest równoważne $r|\varepsilon$ lub innymi słowy: $L(r?) = L(r) \cup \varepsilon$. Operator ? ma ten sam priorytet i łączność co operatory $*$ i +.
3. Wyrażenie regularne $a_1|a_2|...|a_n$, gdzie a_i są symbolami alfabetu, może być zastąpione przez skrót $[a_1a_2...]$. Co więcej, gdy $a_1, a_2, ..., a_n$ tworzą logiczny ciąg, np. kolejne litery duże, litery małe lub cyfry, możemy zastąpić je przez $a_1 - a_n$: $[a - z] = ab...z$

   Przykład: Definicje regularne jak niżej:

```
letter -> A | B |...| Z | a | b |...| z |...
digit -> 0| 1 |...| 9
id -> letter_ (letter_ | digit )*
```

są podstawową formą zapisu.

Definicje regularne możemy zapisać również w skróconej postaci:

```
    letter_  -> [A- Za -z_]
    digit -> [0-9]
    id -> letter_( letter_ | digit )*
```

Rozważmy następujący przykład:

```
stmt -> if expr then stmt
      |  if expr then stmt else stmt
      |  .

expr -> term relop  term
      |  term

term -> id
      | number
```

Wzorce leksemów są przedstawione poniżej:

```
digit -> [0-9]
digits -> digit+
number ->digits (.digits)? ( E [+-]? digits )?
Letter-> [A-Za-z]
id  -> letter ( letter I digit )*
if -> if
then -> then
else -> else
relop -> < | > | < = | > = | = | < >
```

Ponadto możemy przypisać analizatorowi leksykalnemu zadanie rozpoznawania znaków białych określonych przez definicję regularną:

```
ws -> ( blank | tab | newline )+
```

gdzie *blank*, *tab*, i *newline* są to symbole abstrakcyjne, których używamy do wyrażania znaków ASCII o tych samych nazwach.

Token *ws* różni się od innych żetonów tym, że gdy zostaje rozpoznany nie jest przekazywany do parsera, zamiast tego analizator leksykalny przechodzi do rozpoznawania następnego leksemu, który następuje po znaku białym. W tabeli 6.1 przedstawione zostały leksemy i tokeny oraz odpowiadające im wartości atrybutów.

Tabela 6.1. Leksemy, tokeny i atrybuty

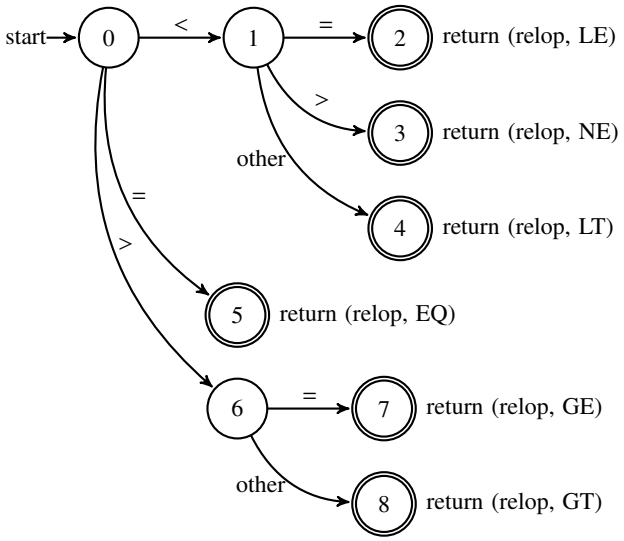
Wyrażenia regularne	Token	Wartość atrybutu
Białe znaki	–	–
if, then, else	if, then, else	–
id, liczba	id, number	wskaźnik do tablicy symboli
<, <=, >, >=, ...	relop	LT, LE, GT, GE, ...

Diagramy przejść (*transition diagrams*). Jako krok pośredni w budowie analizatora leksykalnego, będziemy najpierw konstruować schematy blokowe, nazywane diagramami przejść. Diagramy przejść mają zbiór węzłów, rysowane jako okręgi, i są nazywane stanami. Każdy stan reprezentuje warunek, który może wystąpić podczas procesu skanowania strumienia wejściowego w celu rozpoznawania leksemu, który pasuje do jednego z kilku wzorców. Krawędzie są kierowane z jednego stanu do jakiegoś innego. Każda krawędź jest oznaczona przez symbol lub zbiór symboli. Jeśli jesteśmy w stanie *s* i następnym symbolem jest symbol *a*, to szukamy krawędzi wychodzącej ze stanu *s* i oznakowanej symbolem *a*. Jeśli znajdziemy taką krawędź, to przechodzimy do stanu, do którego prowadzi ta

krawędź. Zakładamy, że wszystkie nasze diagramy przejść są deterministyczne, co oznacza, że taka sama etykieta nie może oznaczać dwóch lub większej liczby krawędzi wychodzących z tego samego stanu.

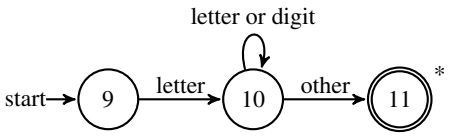
Na rysunku 6.1 pokazany jest diagram przejść dla tokenu relop. Ważne pojęcia związane z diagramami przejść:

1. Stany końcowe, które wskazują, że leksem został rozpoznany; są one oznaczane na rysunkach okręgiem podwójnym. Jeśli ma być wykonana jakaś akcja (zazwyczaj zwrot tokena do parsera), to dołączamy tę akcję do stanu akceptującego.
2. Ponadto, jeśli jest to konieczne, aby przesunąć wskaźnik symbolu o jedną pozycję do przodu (co oznacza, że leksem nie zawiera symbolu, który prowadzi do stanu akceptującego), to należy dodatkowo wstawić znak $*$ obok stanu końcowego.
3. Jeden ze stanów jest nazywany stanem początkowym i oznaczony jest krawędzią wchodzącą o nazwie *start*. Jest to stan diagramu, od którego zaczynamy rozpoznawanie leksemu.

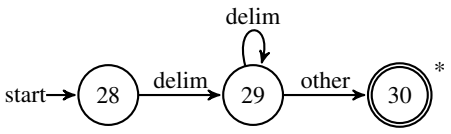


Rys. 6.1. Diagram przejść dla relop

Rysunek 6.2 pokazuje schemat rozpoznawania nazwy, natomiast rysunek 6.3 – schemat rozpoznawania znaków białych.



Rys. 6.2. Rozpoznawanie nazwy



Rys. 6.3. Rozpoznawanie znaków białych

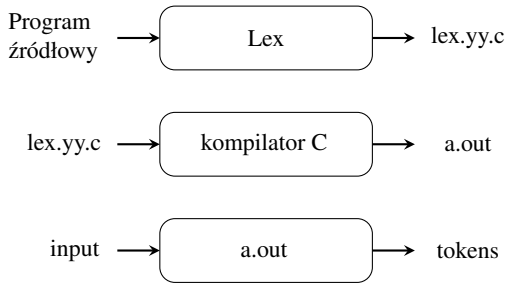
Implementacja analizatora leksykalnego. Możemy wprowadzić zmienną o nazwie *state* do przechowywania bieżącego stanu diagramu przejść. Wtedy możemy zaimplementować analizator leksykalny na podstawie konstrukcji *switch*.

Implementacja diagramu przejść. Każdy stan zawiera odpowiedni segment kodu. Jeśli są jakieś krawędzie wychodzące ze stanu, to jego kod odczytuje jeden znak i wybiera jedną krawędź wychodzącą, jeśli taka istnieje. Używa funkcji *nextchar*(), aby wczytać następny znak z bufora wejściowego.

```
while (1) {
    switch(state) {
        case 0: c=nextchar();
            if (c==blank || c==tab || c==newline){
                state=0; lexeme_beginning++;
            }
            else if (c== '<') state=1;
            else if (c=='=')  state=5;
            else if(c=='>') state=6
            else state=fail();
            break;
        case 9:
            c=nextchar();
            if (isletter(c)) state=10;
            else state=fail();
            break;
    }
}
```

Liczby określające poszczególne stany (zmienna *state*) w powyższym kodzie źródłowym odpowiadają stanom z rysunków 6.1 oraz 6.2.

Rysunek 6.4 pokazuje sposób użycia narzędzia do analizy leksykalnej Lex. Program wejściowy dla narzędzia Lex tworzymy w języku Lex, natomiast samo narzędzie nazywamy kompilatorem Lex. Kompilator Lex konwertuje tokeny wejściowe na diagram przejść oraz generuje kod symulujący diagram przejść, który domyślnie jest zapisywany w pliku o nazwie *lex.yy.c*.



Rys. 6.4. Narzędzie do analizy leksykalnej Lex

Struktura programu w języku Lex jest następująca:

```
    deklaracje
    %%
    reguły translacji
    %%
    funkcje pomocnicze
```

Sekcja deklaracji zawiera deklaracje zmiennych, identyfikatory zadeklarowane jako stałe, np. nazwę tokena oraz definicje regularne. Reguły translacji mają postać: *wzorzec {akcja} (pattern {action})*. Każdy wzorzec jest wyrażeniem regularnym, które może korzystać z definicji regularnych określonych w sekcji deklaracji. Akcje są fragmentami kodu, zwykle napisanego w języku C, choć modyfikacje narzędzia Lex używają innych języków. Trzecia sekcja zawiera dodatkowe funkcje, które są stosowane w akcjach. Alternatywnie funkcje te mogą być kompilowane osobno i ładowane za pomocą analizatora leksykalnego. Wywołany przez parser analizator leksykalny

zaczyna czytać znak po znaku program źródłowy, dopóki nie znajdzie najdłuższego prefiksu, który pasuje do jakiegoś wzorca. Następnie wykonuje działania odpowiedniej akcji. Analizator leksykalny zwraca jedną wartość, nazwę symboliczną, do parsera, ale może w razie potrzeby korzystać ze zmiennej dzielonej o nazwie *yylval* w celu przekazania dodatkowej informacji o znalezionym leksemie.

Rozwiązywanie konfliktów w Lex. Lex stosuje dwie zasady w celu wybrania właściwego leksemu, gdy kilka prefiksów pasuje do jednego lub większej liczby wzorców:

1. Wybiera najdłuższy prefiks.
2. Jeśli najdłuższy prefiks pasuje do dwóch lub więcej wzorców, wybiera wzorzec wcześniejszy w tekście programu Lex.

Przykład: Pierwsza zasada mówi nam, żeby czytać dalej litery i cyfry, chcąc znaleźć najdłuższy prefiks grupy znaków pasujących do identyfikatora. Mówi nam także, żeby potraktować ciąg znaków "<=" jako jeden leksem. Druga zasada sprawia, że rozpoznane słowo jest traktowane jako słowo kluczowe, jeżeli deklarujemy listę słów kluczowych przed deklaracją identyfikatora w programie Lex. Na przykład, jeśli słowo *then* ma najdłuższy prefiks, który pasuje do wzorca, oraz jeżeli *then* występuje wcześniej niż {*id*}, to jest zwracany token THEN (nie ID) .

Przykładowy plik dla programu Lex:

```
1  %{
2  #include <stdio.h>
3  #include "y.tab.h"
4  %}
5
6  %%
7  \=\=          {return EQ;}
8  \!\=          {return NEQ;}
9  \<\=          {return LEQ;}
10 \>\=          {return GEQ;}
11 \<            {return '>';}
12 \>            {return '<';}
13 \(            {return '(';}
14 \)            {return ')';}
15 \!            {return '!';}
16 \=            {return '=';}
17 \+            {return '+';}
18 \-            {return '-';}
19 \*            {return '*';}
20 \/            {return '/';}
21 \;            {return ';';}
22 "int"         {return INT;}
23 "if"          {return IF;}
24 "else"        {return ELSE;}
25 [A-Za-z_][A-Za-z0-9_]*  {return ID;}
26 [1-9][0-9]*|0   {return NUM;}
27 .             {yyerror("Error");
28 %%
```

W powyższym przykładzie następujące linie mają odpowiednie znaczenie. Linia 1 rozpoczyna sekcję nagłówkową, z której kod jest kopiowany bezpośrednio do kodu wyjściowego. Następnie linia 2 dołącza plik nagłówkowy z definicjami funkcji języka C. Linia 3 dołącza plik nagłówkowy zawierający definicje tokenów. Definicje te mogą być wygenerowane przez inny program, np. YACC lub bison, ewentualnie w przypadku prostego analizatora mogą być one zdefiniowane ręcznie. Linia 4 to zakończenie sekcji nagłówkowej. Linia 5 rozpoczyna sekcję z regułami translacji; w następnych liniach są zdefiniowane reguły dopasowania leksemów. Linie od 6 do 11 definiują reguły dla operatorów porównania, natomiast linie 12–17 – leksemy dla operatorów służących do budowania wyrażeń arytmetycznych. Linie 18–20 to przykładowe definicje słów kluczowych języka. Linia 21 definiuje leksem identyfikatora, a linia 22 opisuje leksemy liczb całkowitych. Linia 23 zgłasza błąd w przypadku braku dopasowania znaków na wejściu do wcześniejszych definicji leksemów. Linia 24 kończy tę sekcję; po tej linii może wystąpić

kod w języku C, w tym kod funkcji *main* oraz funkcji pomocniczych (np. *yyerror*). Kod ten zostanie, podobnie jak sekcja nagłówkowa, skopiowany do kodu wyjściowego.

Leksemy zdefiniowane w powyzszym programie pozwalają na przetworzenie prostego języka podobnego do języka C umożliwiającego definiowanie instrukcji warunkowych, prostych wyrażeń i operowania tylko na liczbach całkowitych.

Automaty skończone. Zajmiemy się teraz pytaniem: Co robi Lex z programem wejściowym? Opierając się na programie wejściowym, Lex tworzy automat skończony (*finite automata*).

Automat skończony jest podobny do diagramu przejść z kilkoma różnicami:

1. Automaty skończone służą tylko do rozpoznawania, po prostu zwracają odpowiedź „tak (rozpoznany)" lub „nie (nierozpoznany)" w stosunku do ciągu wejściowego.
2. Automaty skończone dzielimy na:
   a) niedeterministyczne (*nondeterministic finite automata* – NFA), które nie mają ograniczeń na etykiety krawędzi; tym samym symbolem można oznaczyć kilka krawędzi wychodzących z tego samego stanu; napis pusty może być etykietą krawędzi.
   b) deterministyczne (*deterministic finite automata* – DFA), w których każda krawędź, wychodząca z jakiegoś stanu, ma unikatową etykietę.

Zarówno deterministyczne automaty skończone, jak i automaty niedeterministyczne są w stanie rozpoznać te same języki oparte na wyrażeniach regularnych.

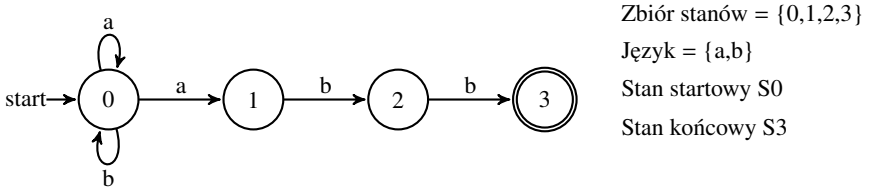
Niedeterministyczny automat skończony zawiera:

1. Skończony zbiór stanów S.
2. Alfabet wejściowy Σ. Zakładamy, że napis pusty ε nie należy do alfabetu Σ.
3. Funkcję przejścia, która określa dla każdego stanu i dla każdego symbolu należącego do zbioru $\Sigma \cup \{\varepsilon\}$ zbiór kolejnych stanów, do których można przejść.
4. Jeden stan początkowy (startowy) $s0$.
5. Zbiór stanów końcowych (akceptowalnych) F, który jest podzbiorem zbioru S.

Możemy reprezentować zarówno NFA, jak i DFA przez graf przejść, w którym węzły reprezentują stany, natomiast krawędzie oznakowane reprezentują funkcję przejścia. Istnieje krawędź oznakowana etykietą a ze stanu s do stanu t wtedy i tylko wtedy, gdy t jest jednym z następnych stanów dla stanu s i wejścia a. Graf ten jest bardzo podobny do diagramu przejść, z następującymi wyjątkami:

– ten sam symbol może oznaczyć kilka krawędzi wychodzących z jednego stanu do kilku różnych stanów;
– krawędź może być oznakowana za pomocą napisu pustego, który oznaczamy jako ε.

Przykładowy niedeterministyczny automat skończony pokazuje rysunek 6.5.



Rys. 6.5. Przykład NFA: rozpoznawanie wyrażenia regularnego $(a|b) * abb$

Tablice przejść. Możemy również reprezentować NFA przez tablicę przejść, której wiersze odpowiadają stanom, natomiast kolumny odpowiadają symbolom wejściowym Σi ε. Wpis dla danego stanu i wejścia jest to wartość zwracana przez funkcję przejścia dla tych argumentów. Jeśli jakiś element tablicy zawiera symbol $\varnothing$, oznacza to, że taki element nie zawiera żadnej informacji. Przykładowa funkcja przejścia została przedstawiona w tabeli 6.2.

Tabela 6.2. **Funkcja przejścia** przedstawiona w formie tabeli

	Wejście	
Stan	a	b
0	0,1	0
1	$\varnothing$	2
2	$\varnothing$	3

NFA akceptuje ciąg znaków x wtedy i tylko wtedy, gdy istnieje jakaś ścieżka w grafie przejść od stanu początkowego do jednego ze stanów końcowych – takiego, że symbole wzdłuż ścieżki tworzą x. Należy pamiętać, że etykiety oznakowane jako ε są ignorowane.

Deterministyczne automaty skończone. Jeśli używamy tablicy przejścia do reprezentowania DFA, to każdy pojedynczy wpis reprezentuje pojedynczy stan; dlatego zapisujemy ten stan bez nawiasów. DFA pozwala na bardzo łatwe rozpoznawanie leksemów. Każde wyrażenie regularne i każdy NFA mogą być konwertowane do DFA akceptującego ten sam język. Na rysunku 6.6 pokazany jest diagram stanów dla $(a|b) * abb$.

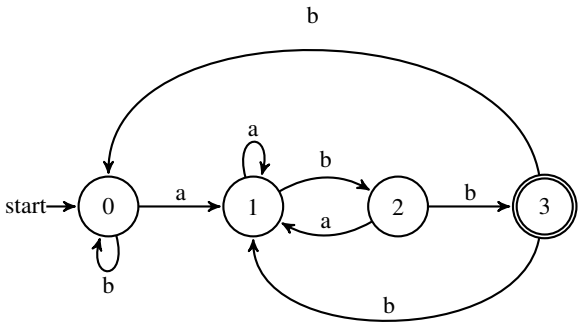
Algorytm: symulacja DFA

Wejście: Napis x, który kończy się znakiem eof. DFA D ze stanem startowym $s0$ i stanami końcowymi F oraz funkcja przejść *move*.

Wyjście: Odpowiedź „tak(*yes*)”, jeśli D rozpoznaje x inaczej „nie(*no*)”.

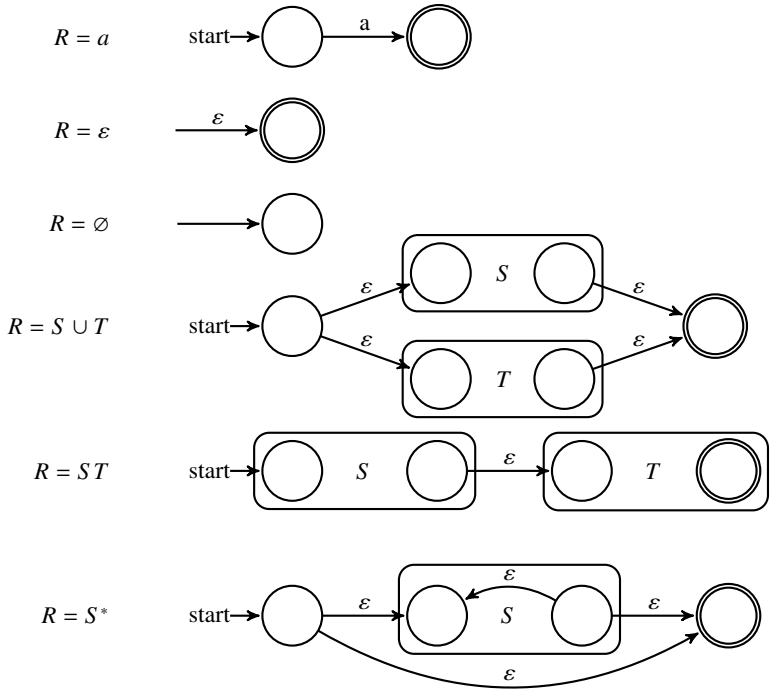
Symulacja DFA:

```
begin

    s := s0;
    c := nextchar;

    while c <> eof do
    begin
        s := move(s, c); // funkcja przejść
        c := nextchar
    end;

    if s is in F then
        return "yes"
    else
        return "no"
    end;

end.
```

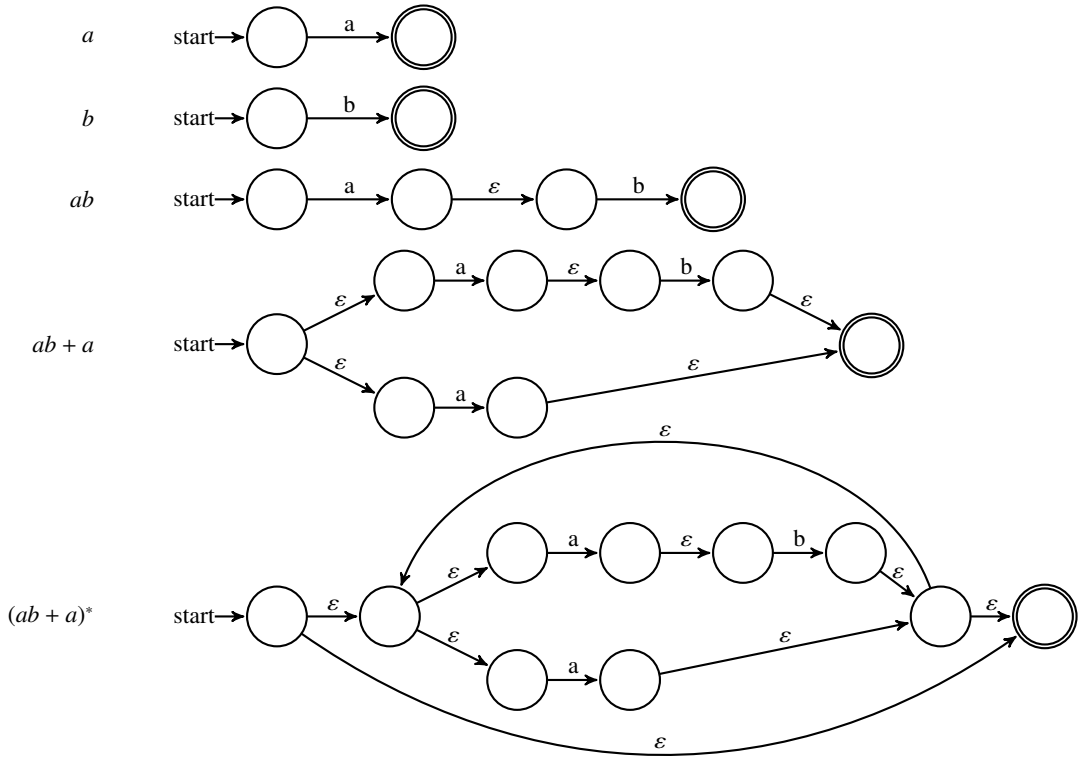


Rys. 6.6. Przykład $(a|b) * abb$

Konwersja RE na NFA. RE zawsze może być przekonwertowane na NFA . W tym celu korzysta się z konwersji bazowych pokazanych na rysunku 6.7. Natomiast na rysunku 6.8 przedstawiono zamianę $R = (ab + a)*$ na NFA. Konwersja rozpoczyna się od najprostszych elementów.



Rys. 6.7. Konwersja RE na NFA – konwersje bazowe



Rys. 6.8. Konwersja RE na NFA – przykład

7. Analiza syntaktyczna 1

Parsowanie. Jest to proces określenia, w jaki sposób ciąg terminali może być generowany przez gramatykę. Dla każdej gramatyki bezkontekstowej istnieje parser, dla którego czas parsowania ciągu z *n* terminali zajmuje co najwyżej $O(n^3)$ czasu. Ale czas ten jest zazwyczaj nie do przyjęcia w praktyce. Na szczęście, dla rzeczywistych języków programowania możemy zaprojektować gramatykę, która może szybko parsować. Dla większości języków programowania zostały opracowane algorytmy parsowania o złożoności liniowej. Analizatory składni języka programowania prawie zawsze przeglądają ciąg wejściowy od lewej do prawej strony, biorąc pod uwagę tylko jeden terminal w danym momencie. Większość metod parsowania należy do jednej z dwóch klas – metod zstępujących (*top-down*) i metod wstępujących (*bottom-up*). Terminy te odnoszą się do kolejności, w jakiej są budowane węzły w drzewie parsowania. W parserach zstępujących (*top-down*), tworzenie drzewa rozpoczyna się od korzenia i polega na przechodzeniu do liści. Natomiast w parserach wstępujących (*bottom-up*) budowa rozpoczyna się od liści i przebiega w kierunku korzenia. Popularność parserów zstępujących jest spowodowana tym, że z łatwością mogą być zbudowane wydajne parsery przy małym nakładzie pracy programisty. Parser wstępujący może obsłużyć jednak więcej bardziej złożonych gramatyk i schematów translacji, co oznacza, że jego zakres stosowalności jest szerszy. W parsowaniu zstępującym budowa drzewa syntaktycznego dokonuje się od korzenia, oznakowanego przez nieterminal startowy; następnie wielokrotnie są wykonywane następujące dwa kroki:

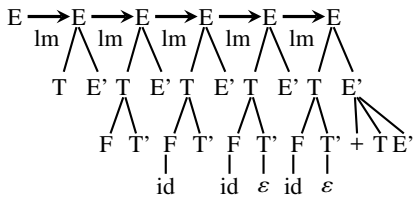
1. W węźle N, oznaczonym nieterminalem A, wybiera się jedną z produkcji, której lewa strona jest A, i tworzy się dzieci dla N oznaczone symbolami prawej strony tej produkcji.
2. Szuka się następnego węzła, w którym poddrzewo ma być utworzone; zazwyczaj jest to skrajny lewy nieterminal drzewa bieżącego.

Aktualny symbol terminalny z ciągu wejściowego jest często nazywany symbolem bieżącym *lookahead*. Początkowo symbolem bieżącym jest pierwszy terminal z lewej strony ciągu wejściowego. Parser próbuje utworzyć drzewo parsujące od korzenia w kierunku liści, skanując wejście od lewej do prawej strony.

Przykład: Dla wejścia $id + id * id$ jest tworzone drzewo parsowania takie jak na rysunku 7.1. Na tym rysunku *lm* oznacza wyprowadzenie lewostronne, czyli zawsze jest wybierany skrajny lewy nieterminal.

Użyte produkcje:

```
1 E -> TE'
2 E' -> +TE' | ε
3 T -> FT'
4 T' -> *FT' | ε
5 F -> (E) | id
```



Rys. 7.1. Parsowanie zstępujące

Parsowanie zstępujące – tworzenie kodu parsera obejmuje kroki:

1. Tworzymy jedną procedurę dla wszystkich symboli terminalnych.
2. Tworzymy jedną procedurę dla każdego symbolu pomocniczego.
3. Tworzymy kod dla każdej produkcji jako ciąg procedur, zgodnie z kolejnością symboli po prawej stronie odpowiedniej produkcji.

Na rysunku 7.2 przedstawione zostało drzewo parsowania dla produkcji:

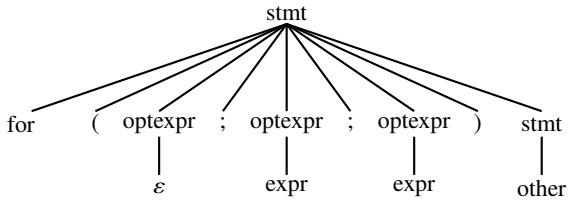
```
stmt->for ( optexpr; optexpr ; optexpr ) stmt
```

dla kodu o przykładowej strukturze:

```
for ( ; i<n ; i+=2*k)
    p=i;
```

Reguły parsowania:

```
stmt -> expr ;
    | if ( expr ) stmt
    | for ( optexpr ; optexpr ; optexpr ) stmt
    | other
optexpr -> ε
    | expr
```



Rys. 7.2. Drzewo parsowania dla produkcji opisujących pętlę *for*

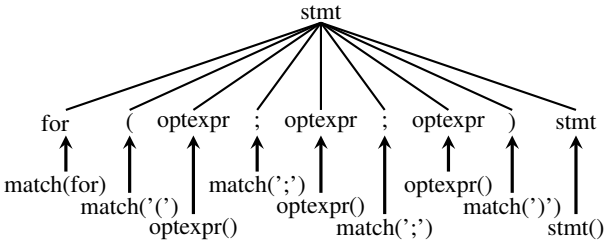
W poniższym kodzie funkcja *stmt* odpowiada za parsowanie elementu *stmt* z wcześniej zaprezentowanych reguł parsowania. Funkcja *optexpr* odpowiada za zastosowanie ε-produkcji, a funkcja *match* odpowiada za sprawdzenie dopasowania terminala *t* do symbolu wejściowego. Pseudokod parsera, który sprawdza czy ciąg wejściowy zawiera błędy syntaktyczne, jest przedstawiony poniżej:

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

Funkcje wykorzystywane w kodzie parsera są przedstawione poniżej:

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

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

Na rysunku 7.3 przedstawione zostało drzewo parsowania wraz z wywołaniami poszczególnych funkcji dla pętli *for*. Natomiast rysunek 7.4 zawiera schemat parsowania zstępującego dla deklaracji tablicy. W momencie wystąpienia pierwszego średnika w konstukcji pętli pojawia się problem braku dopasowania. Wybierana jest wtedy produkcja: $optexpr \rightarrow \varepsilon$. Wywołanie funkcji *stmt* dla *lookahead other* kończy się zakończeniem programu, bez sygnalizacji błędu syntaktycznego, co oznacza, że wejście jest poprawne.



Rys. 7.3. Drzewo parsowania dla produkcji opisujących pętlę *for*

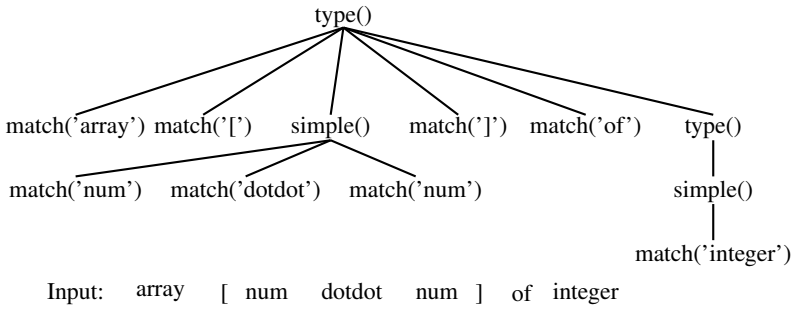
Kroki parsowania: ε-produkcje (produkcje, których prawa strona jest napisem pustym ε) wymagają specjalnego traktowania. Używamy te produkcje jako domyślne, gdy żadna inna produkcja nie może być użyta. Z nieterminalem *optexpr* i symbolem bieżącym: ; ε-produkcja jest wykorzystywana, ponieważ nie ma takiej produkcji, której lewa strona jest *optexpr*, a prawą stroną jest symbol: ;. Ogólnie rzecz biorąc, wybór produkcji dla nieterminala może być oparty na metodzie prób i błędów; to znaczy, że możemy spróbować jakiejś produkcji; jeśli jest ona niewłaściwa, to możemy wrócić i spróbować innej produkcji, itd. Dla gramatyki jak poniżej należy utworzyć odpowiednią liczbę (3) procedur:

```
type -> simple
    | ^ id
    | array [ simple ] of type
simple -> integer
    | char
    | num dotdot num
```

Poniższy program implementuje parsowanie zstępujące:

```
procedure match(t : token);
begin
    if lookahead = t then
        lookahead := nexttoken()
    else
        error()
end;

procedure type();
```

```
begin
    if lookahead in { 'integer', 'char', 'num' } then
        simple()
    else if lookahead = '^' then
        match('^'); match(id)
    else if lookahead = 'array' then
        match('array'); match('['); simple();
        match(']'); match('of'); type()
    else
        error()
end;

procedure simple();
begin
    if lookahead = 'integer' then
        match('integer')
    else if lookahead = 'char' then
        match('char')
    else if lookahead = 'num' then
        match('num');
        match('dotdot');
        match('num')
    else
        error()
end;
```



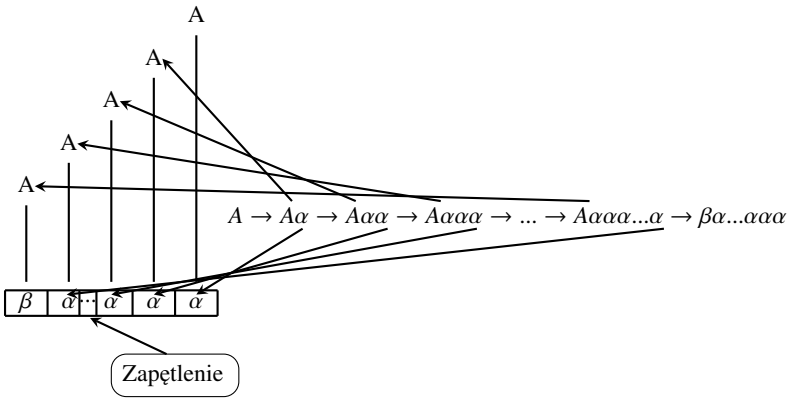
Rys. 7.4. Parsowanie zstępujące dla deklaracji tablicy

Dla parsera zstępującego możliwe jest zapętlenie parsowania. Problem pojawia się, gdy korzystamy z produkcji lewostronnie rekurencyjnych, takich jak: $expr \rightarrow expr + term$, gdzie lewy skrajny symbol ciała produkcji jest taki sam jak symbol po lewej stronie produkcji. Produkcja lewostronnie rekurencyjna może być wyeliminowana poprzez jej modyfikację.

Rozważmy produkcje: $A- > A\alpha|\beta$,
gdzie α i β są to ciągi terminali i nieterminali, które nie zaczynają się od A.

Na przykład dla produkcji: $expr- > expr + term|term$
nieterminal $A = expr$, ciąg $\alpha = +term$, ciąg $\beta = term$.

Nieterminal A i jego produkcja są lewostronnie rekurencyjne. W ogólnym przypadku gramatyka może być lewostronnie rekurencyjna, jeśli nieterminal A wyprowadza napis $A\alpha$ przez zastosowanie dwóch lub większej liczby produkcji bezpośrednich. Powtarzające się stosowanie tej produkcji tworzy sekwencję ciągu α po prawej stronie A. Jeśli w końcu symbol A zostanie zastąpiony przez β, to uzyskamy β, po którym następuje sekwencja zera lub większej liczby α. Na rysunku 7.5 parser zawsze będzie wybierał pierwszą produkcję dla gramatyki: $A- > A\alpha|\beta$. Problem: A nigdy nie zostanie zastąpione przez β, ponieważ zawsze będzie wybrana pierwsza produkcja: $A- > A\alpha$.

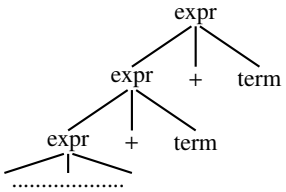


Rys. 7.5. Rekurencja lewostronna – ogólna zasada

Rozważmy następujący przykład gramatyki:

```
expr -> expr + term
      | term
term -> 0,1, ..., 9
```

Dla wejścia: 2 + 2 proces budowania drzewa parsowania dla wyrażenia jest nieskończony, co zobrazowane zostało na rysunku 7.6.

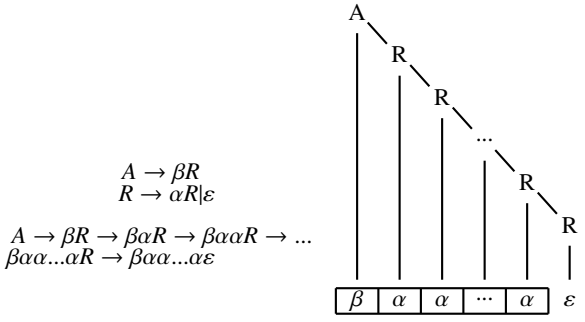


Rys. 7.6. Rekurencja lewostronna – drzewo parsowania

Problem można rozwiązać przez przepisanie produkcji dla A: $A- > A\alpha|\beta$ w następujący sposób przy użyciu nowego nieterminala R.

```
A -> βR
R -> αR
   | ε
```

Rekurencja prawostronna. Nieterminal R i jego produkcja $R- > \alpha R$ są prawostronnie rekurencyjne. Produkcje prawostronnie rekurencyjne prowadzą do drzew, które rosną w dół i w prawo, jak to jest pokazane na rysunku 7.7.



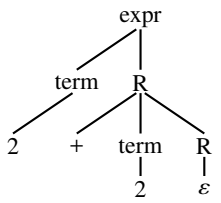
Rys. 7.7. Rekurencja prawostronna – ogólna zasada

Zapis po przekonwertowaniu do powyższej postaci gramatyki z poprzedniego przykładu:

```
expr -> expr + term | term
term->0,1, ..., 9
```

```
expr -> term  R
R -> +term R
   | e
term->0,1,...,9
```

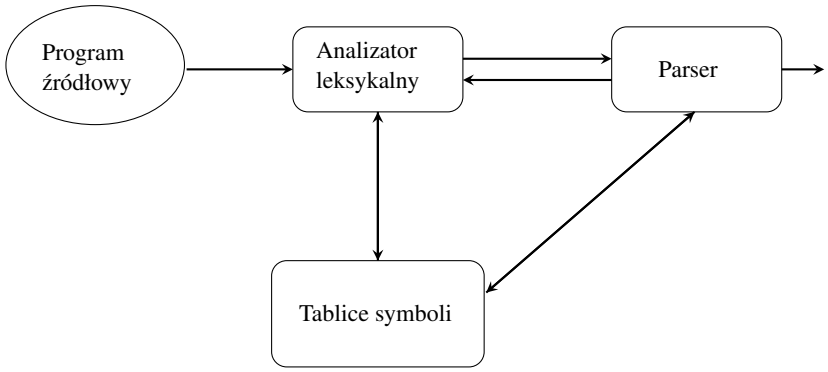
Dla wejścia: 2+2 drzewo parsowania wygląda jak na rysunku 7.8. Liście drzewa na tym rysunku tworzą zdanie wejściowe, więc parser kończy pracę.



Rys. 7.8. Rekurencja prawostronna – drzewo parsowania

8. Analiza syntaktyczna 2

Parser dostaje na wejściu ciąg tokenów od analizatora leksykalnego i sprawdza, czy ciąg ten może zostać wygenerowany przez gramatykę. Miejsce analizatora składniowego w kompilatorze pokazuje rysunek 8.1.



Rys. 8.1. Rola analizatora syntaktycznego

Istnieją trzy główne rodzaje parserów dla gramatyk: uniwersalny, analiza zstępująca(*top-down*) oraz analiza wstępująca (*bottom-up*). Metoda uniwersalna pozwala na zastosowanie dowolnej gramatyki.

Takie ogólne metody jednak mają dużą złożoność obliczeniową i nie mogą być stosowane do tworzenia kompilatorów przemysłowych. Metody stosowane do tworzenia kompilatorów przemysłowych są oparte na analizie zstępującej (*top-down*) oraz analizie wstępującej (*bottom-up*).

Wyprowadzenia. Rozważmy następującą gramatykę:
$E \Rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$

Zamianę symbolu E na $-E$ zapisujemy jako: $E \Rightarrow -E$ i czytamy: „E wyprowadza $-E$."

Inne przykłady wyprowadzeń: $E * E \Rightarrow (E) * E$ lub $E * E \Rightarrow E * (E)$

Możemy wziąć pojedynczy symbol E i wielokrotnie stosować produkcje w dowolnej kolejności. Na przykład: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$.

Załóżmy, że nieterminal A znajduje się w środku napisu: $\alpha A \beta$, gdzie α i β są to dowolne podnapisy. Przypuśćmy, że $A \Rightarrow \gamma$ jest to produkcja, wtedy wyprowadzenie ma postać: $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Symbol: $\Rightarrow$ oznacza wyprowadzenie bezpośrednie.

Gdy mamy sekwencję wyprowadzeń: $a_1 \Rightarrow a_2 \Rightarrow ... \Rightarrow a_n$, mówimy, że a_1 wyprowadza a_n co można zapisać również jako $a_1 \Rightarrow^* a_n$, gdzie symbol $\Rightarrow^*$ oznacza „zero wyprowadzeń lub więcej wyprowadzeń". Będziemy stosować również symbol $\Rightarrow^+$, który oznacza „co najmniej jedno wyprowadzenie lub więcej wyprowadzeń". Jeśli $S \Rightarrow^+ a$, gdzie S jest to symbol startowy w gramatyce G, to mówimy, że a jest to forma zdaniowa gramatyki G.

Uwaga: Forma zdaniowa może zawierać jednocześnie symbole terminalne jak i nieterminalne.

Zdaniem gramatyki G jest forma zdaniowa, której wszystkie symbole są terminalami. Język generowany przez gramatykę G jest to zbiór wszystkich zdań generowanych przez G.

Rodzaje wyprowadzeń mogą być następujące:

1. W wyprowadzeniach lewostronnych (*leftmost derivations*), zawsze wybieramy do zamiany pierwszy nieterminal z lewej strony formy zdaniowej. Jeśli $\alpha \Rightarrow \beta$ w kroku, w którym zamieniany jest skrajnie lewy nieterminal z α, to wyprowadzenie takie zapisujemy: $\alpha \Rightarrow_{lm} \beta$.
2. W wyprowadzeniach prawostronnych (*rightmost derivations*), zawsze wybieramy do zamiany pierwszy nieterminal z prawej strony formy zdaniowej. Jeśli $\alpha \Rightarrow \beta$ w kroku, w którym zamieniany jest skrajnie prawy nieterminal z α, to wyprowadzenie takie zapisujemy: $\alpha \Rightarrow_{rm} \beta$.

Zbiór First. Zbiór FIRST(α), gdzie α jest dowolnym ciągiem symboli gramatycznych, jest to zbiór terminali, od których zaczynają się ciągi wyprowadzane z α. Jeśli $\alpha \Rightarrow^* \varepsilon$, to α należy do FIRST(α). Dla poniższej gramatyki, druga kolumna tabeli 8.1 pokazuje kilka elementów zbioru FIRST.

```
exp -> term exp'
exp' -> addop term exp' | ε
addop -> + | -
term -> factor term'
term' -> mulop factor term' | ε
mulop -> *
factor -> ( exp ) | num
```

Tabela 8.1. Zbiór First – przykład

Produkcja	First	Numer produkcji
exp	(,...	1
exp'	ε ,...	2
addop	+,-,...	3
term	(,num,...	4
term'	ε,...	5
mulop	*,...	6
factor	(,num,...	7

Zbiór FOLLOW. Zbiór FOLLOW(A) dla nieterminala A jest to zbiór terminali a, które mogą wystąpić bezpośrednio na prawo od A w pewnej formie zdaniowej. Ponadto jeśli A jest skrajnym symbolem w formie zdaniowej, to dodajemy symbol $ do zbioru FOLLOW(A), gdzie $ jest to symbol specjalny oznaczający koniec napisu (*endmarker*) i nienależący do żadnej gramatyki. Dla poniższej gramatyki trzecia kolumna tabeli 8.2 pokazuje kilka elementów zbioru FOLLOW.

```
exp -> term exp'
exp' -> addop term exp' | ε
addop -> + | -
term -> factor term'
term' -> mulop factor term' | ε
mulop -> *
factor -> ( exp ) | num
```

Tabela 8.2. Zbiór First i Follow – przykład

Produkcja	First	Follow	Numer produkcji
exp	(num	$)	1
exp'	ε + -	$)	2
addop	+ -		3
term	(num	+ - $)	4
term'	ε *	)	5
mulop	*		6
factor	(num		7

Gramatyki LL(1). Pierwsze „L” w LL(1) oznacza skanowanie wejścia od lewej do prawej strony, drugie „L” – zastosowanie wyprowadzenia lewostronnego, a „1” – że w każdym kroku parsowania tylko jeden symbol wejściowy jest brany pod uwagę. Klasa gramatyk LL(1) jest dość bogata, aby mogła uwzględnić większość konstrukcji programistycznych. Ale są również ograniczenia – gramatyki rekurencyjne lewostronnie oraz gramatyki niejednoznaczne nie należą do gramatyk LL(1). Gramatyka G należy do LL(1) wtedy i tylko wtedy, gdy dla dwóch różnych produkcji $A \Rightarrow \alpha|\beta$, należących do G, są spełnione następujące warunki:

1. Dla każdego terminala a, z α i β nie daje się jednocześnie wyprowadzić ciągu rozpoczynającego się od a.
2. Co najwyżej z jednego α i β daje się wyprowadzić napis pusty.
3. Jeśli $\beta \Rightarrow \varepsilon$, to α nie wyprowadza napisu, którego pierwszy symbol jest terminalem i który znajduje się w zbiorze FOLLOW(A). Jeśli $\alpha \Rightarrow \varepsilon$, to β nie wyprowadza napisu, którego pierwszy symbol jest terminalem i który znajduje się w zbiorze FOLLOW(A).

Pierwsze dwa warunki oznaczają, że zbiory FIRST(α) i FIRST(β) są rozłączne. Trzeci warunek oznacza, że jeśli ε należy do zbioru FIRST(β), to zbiory FIRST(α) i FOLLOW(A) są rozłączne; podobnie, gdy ε należy do zbioru FIRST(α).

Tworzenie tablicy do parsowania przewidującego. Parsowanie z wykorzystaniem gramatyk LL(1) wymaga w pierwszej kolejności utworzenia tablicy parsowania. Do tworzenia tablic parsowania są stosowane następujące reguły:

1. Dla każdej produkcji $A \Rightarrow \alpha$ z gramatyki wykonuje się kroki podane niżej.
2. Dla każdego terminala a z FIRST(α) dodaje się produkcję $A \Rightarrow \alpha$ do macierzy $M[A, a]$.
3. Jeśli ε należy do FIRST(α), to dla każdego terminala b z FOLLOW(A), dodaje się $A \Rightarrow \alpha$ do macierzy $M[A, b]$.
4. Jeśli ε należy do FIRST(α) oraz \$ należy do FOLLOW(A), to dodaje się $A \Rightarrow \alpha$ do $M[A, \$]$.
5. Jeśli jakieś pole tablicy jest puste po wykonaniu kroków 2–4, to oznacza to, że jeśli zostanie wybrane takie pole, to analizator zwraca błąd i kończy parsowanie.

W tabeli 8.3 przedstawiony jest schemat tworzenia tablicy parsowania dla poniższych zbiorów FIRST i FOLLOW i przedstawionej poniżej gramatyki:

```
1              First           Follow
2  exp         {(, num}        {$,)}
3  exp'        {+,-, ε}        {$,)}
4  addop       {+,-}           {(, num}
5  term        {(, num}        {+,-,),$}
6  term'       {*, ε}          {+,-,),$}
7  mulop       {*}             {(, num}
8  factor      {(, num}        {*,+,-,),$}
```

```
1  exp -> term exp'
2  exp' -> addop term exp'
3  exp' -> ε
4  addop -> +
5  addop -> -
6  term -> factor term'
7  term' -> mulop factor term'
8  term' -> ε
9  mulop -> *
10 factor -> ( exp )
11 factor -> num
```

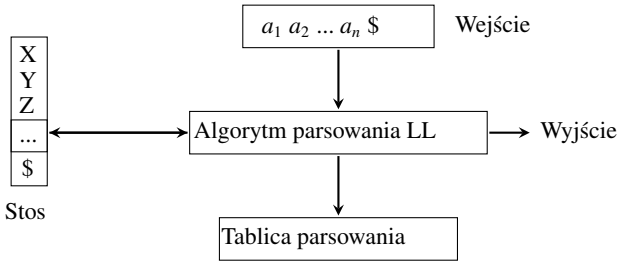
Dla każdej gramatyki LL(1) każde pole(wpis) w tablicy parsowania ma tylko jedną produkcję lub jest puste (co oznacza błąd). Jeśli dla danej gramatyki jakieś pole tablicy zawiera dwie lub większą liczbę produkcji, to taka gramatyka nie należy do klasy LL(1). Na rysunku 8.2 pokazana jest architektura parsera LL(1).

Na rysunku 8.3 pokazana jest zasada działania stosu. Algorytm parsowania bazuje na poniższych operacjach na stosie:

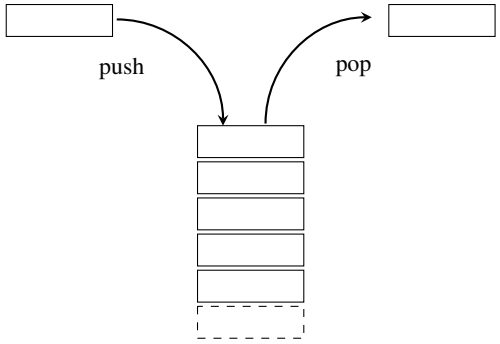
– *push* – odłożenie obiektu na stos,
– *pop* – ściągnięcie obiektu ze stosu i zwrócenie jego wartości.

Tabela 8.3. Tworzenie tablicy parsowania

	(	)	+	-	*	n	$
exp	1					1	
exp'		3	2	2			3
addop			4	5			
term	6					6	
term'		8	8	8	7		8
mulop					9		
factor	10					11	



Rys. 8.2. Architektura parsera LL(1)



Rys. 8.3. Operacje na stosie

Załóżmy, że zmienna o nazwie *top* przechowuje zawartość, która znajduje się na szczycie stosu (pierwsza wartość do odczytu); zmienna o nazwie *input* przechowuje bieżący terminal zdania wejściowego; *M[top, input]* oznacza zawartość wpisu tablicy parsowania dla argumentów *top* i *input*. Wtedy algorytm parsowania wygląda jak poniżej:

```
1  odłóż na stos dwa symbole: '$ Symbol Startowy'
2  ($ oznacza koniec napisu).
3  1) if top == input == $ then
4       akceptuj - zdanie wejściowe jest poprawne;
5  2) if top == input then
6       zdejmij wartość ze szczytu stosu;
7       odczytaj następny symbol wejściowy i podstaw go pod zmienną input;
8       goto 1;
9  3) If top is nonterminal then
10      if M[top, input] jest to produkcja then
11         zastąp wartość na szczycie stosu przez ciało tej produkcji;
12         goto 1;
13      else error // zawartość M[top, input] jest polem pustym
14    Else
15      error //top zawiera terminal
```

Wyprowadzenia z poniższej listy mają numery, które zostały użyte w tabeli 8.4.

```
1 E->TE'
2 E'->+TE'
3 E'-> ε
4 T->FT'
5 T'->*FT'
6 T'-> ε
7 F->(E)
8 F->id
```

Tabela 8.4. Parsowanie – przykład

	id	+	*	(	)	$
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(1)			(1)		

Tabela 8.5. Kroki parsowania

Stos	Wejście	Produkcja
$E	id+id*id$	
$E'T	id+id*id$	E->TE'
$E'T'F	id+id*id$	T->FT'
$E'T'id	id+id*id$	F->id
$E'	+id*id$	T'->ε
$E'T+	+id*id$	E'->+TE'

Dla kroków przedstawionych w tabeli 8.5 uzyskujemy następujące wyprowadzenie lewostronne:

$$E => TE' => FT'E' => idT'E' => id + TE$$

a następnie:

$$E => TE' => FT'E' => idT'E' => .... => id + id * id$$

9. Analiza wstępująca 1

Analiza wstępująca jest bardziej ogólna niż zstępująca. Jest preferowana w praktyce, ale jest bardziej złożona od analizy zstępującej. Dalej będziemy korzystać z następującej gramatyki:

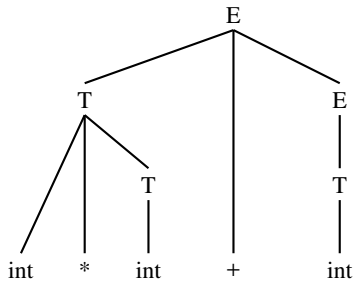
```
1 E -> T + E | T
2 T -> int * T | int | (E)
```

Redukcja jest to krok odwrotny do wyprowadzenia. Rozważmy zdanie: $int*int+int$. Po skorzystaniu z produkcji $T \Rightarrow int$, wynik redukcji jest następujący: $int * T + int$.

Analiza wstępująca redukuje zdanie do symbolu startowego. Analiza wstępująca zmierza w kierunku od zdania do symbolu startowego, czyli w poniższym fragmencie gramatyki od linii 1 do linii 6.

```
1 int * int + int      T -> int
2 int * T + int        T -> int * T
3 T + int              T -> int
4 T + T                E -> T
5 T + E                E -> T + E
6 E
```

Jeśli zastosujemy produkcje wykorzystane przez analizę wstępującą w kolejności odwrotnej, to uzyskamy wyprowadzenie prawostronne. Schemat analizy zstępującej dla powyższych produkcji jest zaprezentowany na rysunku 9.1.



Rys. 9.1. Analiza wstępująca

W poniższym pseudokodzie został przedstawiony algorytm parsowania; jego wejściem jest zdanie I:

```
1 Powtarzaj
2     w zdaniu I wybierz niepusty podciąg,
3     gdzie β jest prawą stroną produkcji X->β;
4     w zdaniu I zastąp β przez X;
5 Dopóki I!="S" (symbol startowy) lub wszystkie możliwości są wyczerpane.
```

Pytania:
- Jak wybrać podciąg w każdym kroku?
- Czy ten algorytm zawsze się kończy?
- Jaka jest jego złożoność?
- Czy obsługuje dowolną gramatykę?

Ważne wnioskowanie dotyczące analizy wstępującej:
- Niech $\alpha\beta\omega$ będzie napisem bieżącym po jakiejś liczbie kroków analizy wstępującej.
- Załóżmy, że następna redukcja korzysta z produkcji $X \Rightarrow \beta$.
- Wtedy ω jest to podciąg terminali.

Dlaczego? Dlatego, że $\alpha X\omega \Rightarrow \alpha\beta\omega$ jest to krok wyprowadzenia prawostronnego, czyli ω nie może zawierać żadnego nieterminala.

Idea parsowania – należy podzielić napis na dwa podciągi:
- prawy podciąg, który jest jeszcze niezbadany przez parser (ciąg terminali);
- lewy podciąg, który zawiera tylko nieterminale.

Punkt podziału jest oznaczony symbolem |, który nie jest częścią napisu.

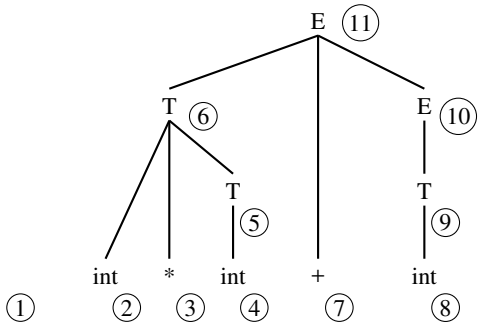
Parsowanie *shift-reduce*. Analiza wstępująca obejmuje dwie akcje:
- przesunięcie (*shift*),
- redukcja (*reduce*).

Shift: Przesuń symbol: | o jedno miejsce w prawo. Przesuwa terminal do lewego podciągu. $ABC|xyz\ ABCx|yz$
Reduce: Niech $A \Rightarrow xy$ będzie produkcją. *Reduce*: Zastąp podciąg xy (prawa strona produkcji) przez A (lewa strona produkcji).

Przykład: $Cbxy|ijk\ \ CbA|ijk$.

Na rysunku 9.2 za pomocą liczb w okręgach przedstawione zostały kolejne kroki parsowania *shift-reduce* z poniższego schematu:

```
1  |int * int + int          shift
2  int | * int + int         shift
3  int * | int + int         shift
4  int * int | + int         reduce T -> int
5  int * T | + int           reduce T -> int * T
6  T | + int                 shift
7  T + | int                 shift
8  T + int |                 reduce T -> int
9  T + T |                   reduce E -> T
10 T + E |                   reduce E -> T + E
11 E |
```



Rys. 9.2. Parsowanie *shift-reduce*

Zastosowanie stosu. Lewy ciąg może być obsługiwany za pomocą stosu:
- przed parsowaniem na szczyt stosu dodajemy symbol |,
- *shift:* dodaje terminal na stos,

– *reduce:* zdejmuje 0 lub większą liczbę symboli (prawa strona zastosowanej produkcji) ze stosu i dodaje nieterminal na stos (lewa strona zastosowanej produkcji).

Kiedy kompilator korzysta z *shift*, kiedy z *reduce*? Rozważmy napis: $int * int + int$. Stosując $T \Rightarrow int$, możemy zredukować napis jak wyżej do $T|*int+int$. Fatalny błąd: Nigdy w ten sposób nie zredukujemy zdania do symbolu startowego E.

Zastosowanie stosu – strategia wyboru akcji:

– jeśli uchwyt jest na szczycie stosu, to wykonaj *reduce*;
– jeżeli jest inaczej, wykonaj *shift*.

„Uchwyt" jest to podciąg pasujący do prawej strony jakiejś produkcji. Podstawowe jest pytanie: Jak w sposób formalny rozpoznać „uchwyt"? Zostało opracowanych wiele metod do rozpoznania „uchwytu" w sposób formalny. Wiele metod daje nam wiele gramatyk.

Gramatyki (i ich parsery) można sklasyfikować następująco:

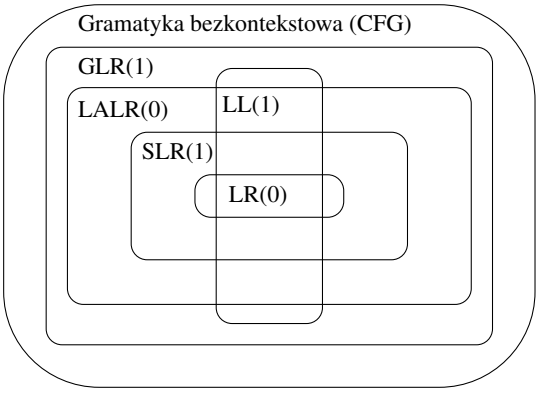
Parser LR (k) „L" oznacza przeglądanie wejścia od lewej do prawej, „R" oznacza budowę zdania wejściowego przez zastosowanie wyprowadzenia prawostronnego na podstawie produkcji zwróconych przez parser w kolejności od ostatniej do pierwszej, k oznacza liczbę symboli podglądanych podczas analizy. Im większa jest wartość k, tym szerszy jest zakres stosowalności parsera – może on parsować więcej gramatyk. Ale jednocześnie ze wzrostem k rośnie złożoność procesu tworzenia parsera oraz wydłuża się czas parsowania. Można zbudować analizatory LR do prawie wszystkich konstrukcji z języków programowania. Jest wiele modyfikacji parserów LR(k). Każda modyfikacja ma taką samą architekturę parsera i taki sam jest sposób jego działania. Jedyną różnicą jest sposób tworzenia tablic parsowania. Dalej rozważymy najbardziej popularne modyfikacje parsera LR(k).

Parser SLR (k) – *Simple* LR(k) parser – prosty LR(k) parser. Jest on najsłabszy ze względu na liczbę gramatyk, dla których działa, ale jest najprostszy w implementacji.

LALR (1) – *LookAhead* LR(1) – podglądający LR(1) parser. Tablice uzyskiwane przy jego zastosowaniu są znacznie mniejsze niż tablice LR(1). Narzędzia YACC i bison są oparte na LALR(1).

Parser GLR – *Generalized* (uogólniony) LR parser jest najbardziej zaawansowaną modyfikacją parsera LR; może parsować prawie każdą gramatykę.

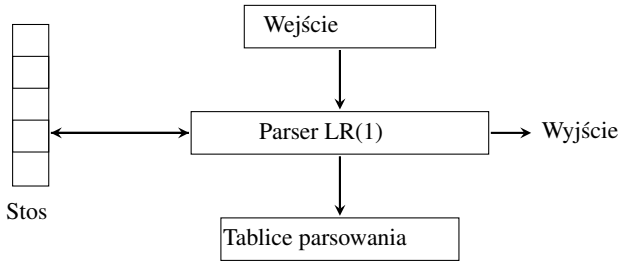
Na rysunku 9.3 przedstawiającym gramatyki im większa jest powierzchnia, zaznaczona dla jakiejś gramatyki, tym większy jest zakres jej zastosowania.



Rys. 9.3. Gramatyki

10. Analiza wstępująca 2

Parser LR(1). Parsowanie jest wykonywane na podstawie automatu skończonego. Parsowanie jest niezależne od języka. Automat skończony jest generowany na podstawie gramatyki wejściowej. Na rysunku 10.1 pokazana jest architektura parsera LR(1).



Rys. 10.1. Ogólna architektura parsera LR(1)

Automat PDA. Automat skończony, który korzysta ze stosu, jest nazywany automatem PDA (*pushdown automaton*). Działanie PDA jest określone przez jego obecny stan przechowywany na szczycie stosu oraz bieżący symbol wejściowy.

Parser LR (1) wykorzystuje tablicę parsowania, bufor wejściowy i stos stanów. Wykonuje trzy akcje:
- przesuwa (*shift*) token z bufora wejściowego na stos,
- redukuje (*reduce*) zawartość stosu przez zastosowanie produkcji,
- przechodzi do nowego stanu (*go to*).

Sytuacje LR(0). Żeby zbudować tablicę parsowania, musimy najpierw znaleźć sytuacje (*items*) LR(0). Sytuacją LR(0) nazywamy produkcję z kropką (•) w jakimś miejscu jej prawej strony. Dla produkcji $E->E+T$ sytuacje LR(0) są takie jak poniżej:

```
1 E -> • E + T
2 E -> E • + T
3 E -> E + • T
4 E -> E + T •
```

Kropka (•) znajduje się we wszystkich możliwych miejscach prawej strony produkcji.

Interpretacja sytuacji $A \Rightarrow \alpha \bullet \beta$: „Napis α już został przetworzony, więc możemy przetwarzać β". To, czy rzeczywiście będziemy przetwarzać β, jest uzależnione od kolejnych symboli wejścia. Zbiór sytuacji LR(0) reprezentuje pojedynczy stan PDA. W parsowaniu LR musimy wzbogacić gramatykę przez dodanie produkcji: $S'->S$, gdzie S' jest to nowy symbol startowy. Gramatyka wzbogacona gwarantuje to, że nowy symbol startowy nie spowoduje zapętlenia algorytmu parsowania.

Stany PDA. Stan początkowy nazywa się I0 (sytuacja 0). Stan I0 jest to domknięcie zbioru zawierającego jedną sytuację: $\{S' \bullet S\}$.

Sposób obliczenia domknięcia zbioru sytuacji:

– Dla każdej sytuacji $A- > \alpha \bullet B\beta$ w zbiorze oraz dla każdej produkcji $B \Rightarrow \gamma$ gramatyki dodaje się sytuację $B \Rightarrow \bullet\gamma$ do zbioru,
– Wykonywanie jest kontynuowane, dopóki nie będzie żadnych nowych elementów do dodania do zbioru.

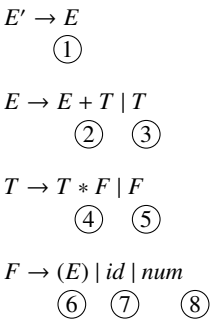
Przykład dla gramatyki:

```
1  E' -> E
2  E -> E + T | T
3  T -> T * F | F
4  F -> (E) | id | num
```

Stan I_0 zawiera sytuacje, które należą do domknięcia sytuacji $E'- > \bullet E$:

```
1  E' -> • E
2  E -> • E + T
3  E -> • T
4  T -> • T * F
5  T -> • F
6  F -> • (E)
7  F -> • id
8  F -> • num
```

Na rysunku 10.2 reprezentującym domknięcia dla powyższych produkcji liczby w okręgach oznaczają numery z powyższej listy.

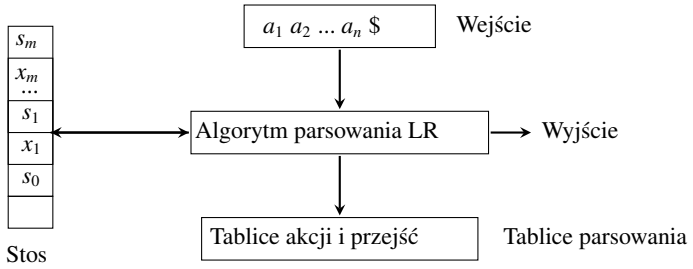


Rys. 10.2. Domknięcia – objaśnienie

Jeśli $A- > \alpha \bullet X\beta$ jest to sytuacja opisująca (razem z innymi sytuacjami) jakiś stan, to:

– przejście z tego stanu może nastąpić, gdy symbol X jest przetwarzany,
– przejście następuje do stanu, który jest reprezentowany domknięciem sytuacji $A- > \alpha X \bullet \beta$.

Na przykład dla E ze zbioru sytuacji $E'- > \bullet E, E- > \bullet E + T$ mamy przejścia do $E'- > E\bullet, E- > E \bullet +T$.



Rys. 10.3. Architektura parsera LR(k)

Na rysunku 10.3 pokazana jest architektura parsera LR(k). Dla każdej modyfikacji parsera LR(k) zostały opracowane algorytmy tworzenia tablic parsowania, które można znaleźć w książce: „*Compilers, Principles, Techniques, and Tools*"[1].

Tablica parsowania ma dwie części – tablicę akcji i tablicę przejść. Wpisy w tablicy akcji $[s,t]$ mogą mieć cztery wartości:

- przesunięcie (*shift*) s_i, gdzie s_i oznacza stan, do którego należy przejść;
- redukcję (*reduce*) z zastosowaniem produkcji (zazwyczaj podany jest numer produkcji);
- akceptację (*accept*);
- błąd (*error*).

Wpis w tablicy przejść (*go to*) $[s,T] = s_i$ oznacza przejście ze stanu s i symbolu nieterminalnego T do stanu s_i. Z tablicy przejść korzystamy po każdej redukcji – uwzględniamy bieżący stan oraz symbol na szczycie stosu (lewa strona zastosowanej produkcji) jako wejście do tablicy i dodajemy stan, który zwraca tablica, na szczyt stosu.

Gramatyki LR(k). Gramatyka, która pozwala na parsowanie za pomocą analizatora LR, podglądającego co najwyżej k symboli wejściowych, nazywana jest gramatyką LR(k).

Konflikty w parserach LR(k). O konflikcie mówimy wtedy, gdy jakiś wpis w tablicy parsowania zawiera dwie lub większą liczbę akcji: przesunięcie / redukcja lub redukcja / redukcja. Konflikty mogą być rozwiązywane automatycznie, jeśli korzystamy z narzędzi do tworzenia parserów, na przykład YACC, lub mogą wymagać ingerencji twórcy parsera w celu wskazania akcji właściwej.

Akcje parsera LR(1). Parser LR(1) korzysta z następujących akcji:

```
1  if akcja[ s, a] = shift s' then
2      wstaw a, a następnie s' na szczyt stosu,
3      przejdź do następnego symbolu  wejściowego;
4  if akcja [s, a] = reduce (A->β) then
5      zdejmij ze stosu 2*|β| symboli;
6      niech s' będzie stanem, który znalazł się na wierzchołku
7      stosu (po usunięciu  ze stosu 2*|β| symboli);
8      wstaw A  i wartość, którą zwraca tablica przejść (s', A),
9      na wierzchołek stosu;
10     przekaż na wyjście produkcję A->β  ;
11 if akcja[s, a] = accept then
12     koniec;
13 else
14     błąd().
```

gdzie $|\beta|$ oznacza długość napisu $|\beta|$.

Rysunek 10.4 i tabele 10.1 i 10.2 pokazują przykład sposobu działania parsera LR(1) dla następującej gramatyki:

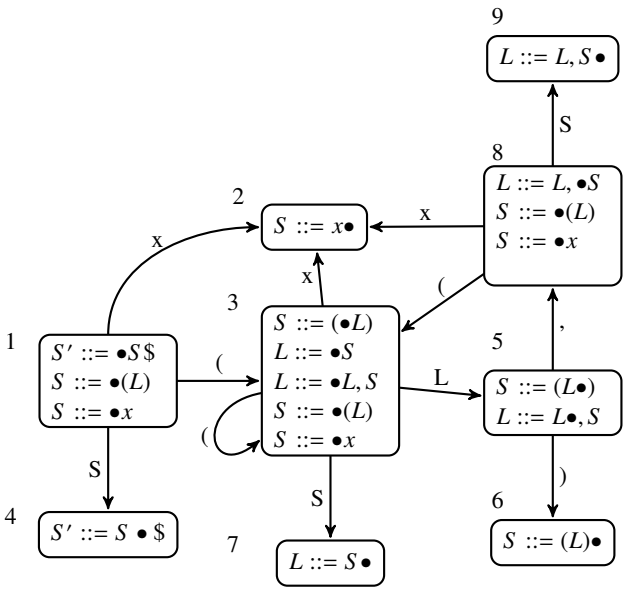
```
0  S' -> S \$
1  S -> ( L )
2  S -> x
3  L -> S
4  L -> L , S
```

dla której symbolami terminalnymi są : '(', ')', ',', 'x', natomiast symbolami nieterminalnymi są L i S.

Uwaga: Znak przecinka (np. ten z 4 linii powyższej listy produkcji) i znaki nawiasów (linia numer 1) są pełnoprawnymi terminalami. Dla wejścia (x,x)\$ parser wykona następujące akcje:

1 s_3 s_2 r_2g_7 r_3g_5 s_8 s_2 r_2g_9 r_4g_5 s_6 r_1g_4 *accept*

gdzie s_i oznacza *shift* do stanu i, r_ig_j oznacza *reduce* za pomocą produkcji i i przejście do stanu j; 1 jest stanem startowym. Komórki tabeli 10.1 w indeksie górnym wskazują numer kroku z tabeli 10.2.



Rys. 10.4. Automat skończony

Tabela 10.1. Kroki parsera LR(1)

Stan	(	)	x	,	$	S	L
1	s_3 [1]		s_2			g_4 [15]	
2	r_2	r_2	r_2 [9]	r_2 [3]	r_2		
3	s_3		s_2 [2]			g_7 [4]	g_5 [6 12]
4					a [16]		
5		s_6 [13]		s_8 [7]			
6	r_1	r_1	r_1	r_1	r_1 [14]		
7	r_3	r_3	r_3	r_3 [5]	r_3		
8	s_3		s_2 [8]			g_9 [10]	
9	r_4	r_4 [11]	r_4	r_4	r_4		

Tabela 10.2. Kroki parsera LR(1) – stan stosu i wejścia

Krok	Wejście	Stos	Akcja
1	|(x,x)$	1	s_3
2	(|x,x)$	1(3	s_2
3	(x|,x)$	1(3x2	r_2
4	(x|,x)$	1(3S	g_7
5	(x|,x)$	1(3S7	r_3
6	(x|,x)$	1(3L	g_5
7	(x|,x)$	1(3L5	s_8
8	(x,|x)$	1(3L5,8	s_2
9	(x,x|)$	1(3L5,8x2	r_2
10	(x,x|)$	1(3L5,8S	g_9
11	(x,x|)$	1(3L5,8S9	r_4
12	(x,x|)$	1(3L	g_5
13	(x,x|)$	1(3L5	s_6
14	(x,x)|$	1(3L5)6	r_1
15	(x,x)|$	1S	r_1
16	(x,x)$ |	1S4	accept

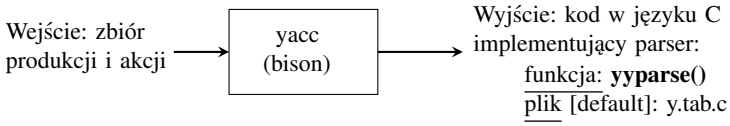
11. Narzędzie YACC

YACC. Jest to generator analizatorów składniowych. Autorem oryginalnej wersji YACC jest Stephen C. Johnson, 1975.

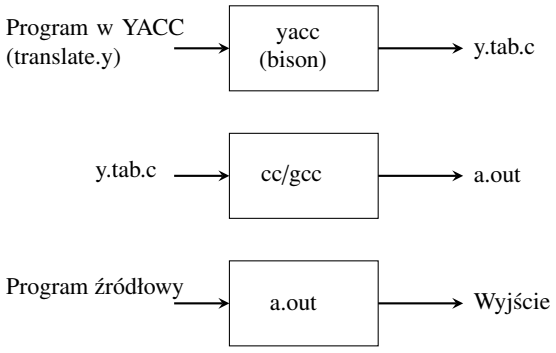
Narzędzia pokrewne:
- lex, yacc (AT&T),
- bison (GNU),
- BSD yacc,
- PCYACC (Abraxas Software).

Zadaniem generatora YACC jest wygenerowanie kodu źródłowego analizatora składniowego w języku C. Kod źródłowy generowany jest przez YACC na podstawie pliku ze specyfikacją, co pokazane jest na rysunku 11.1. Natomiast na rysunku 11.2 pokazany jest sposób korzystania z narzędzia YACC. Sposób współpracy kodu przygotowanego za pomocą YACC z analizatorem leksykalnym został zaprezentowany na rysunku 11.3.

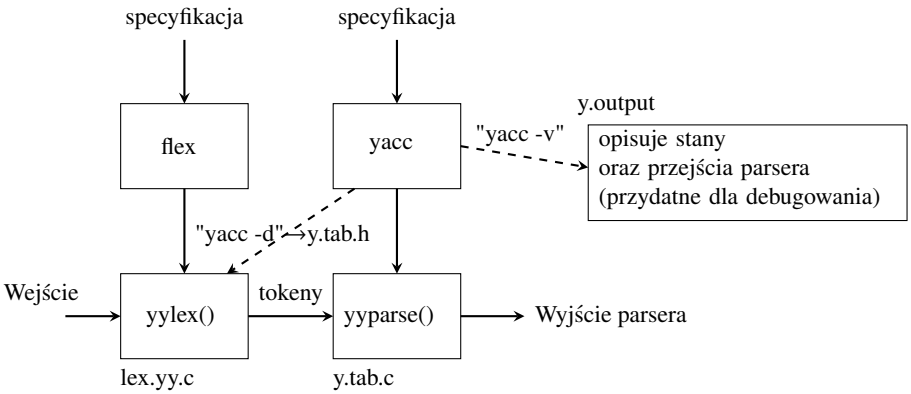
Generator YACC wykonuje następujące czynności:
- pobiera specyfikację gramatyki bezkontekstowej,
- generuje kod dla parsera.



Rys. 11.1. Generator YACC



Rys. 11.2. Jak korzystamy z YACC

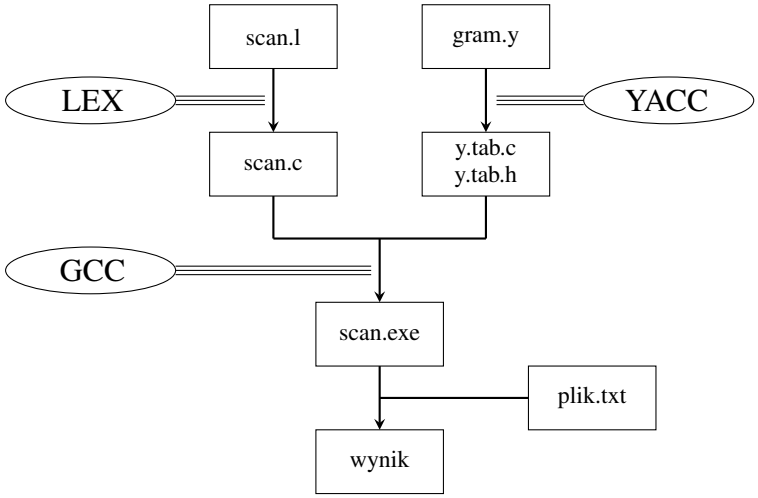


Rys. 11.3. Zastosowanie YACC

Zasady działania programu YACC są następujące:

- Wygenerowany przez program YACC analizator redukujący działa na podstawie tablicy LALR(1).
- Jako symbol startowy gramatyki przyjmowany jest, przez domniemanie, nieterminal znajdujący się po lewej stronie pierwszej produkcji.
- Wygenerowany parser ma postać funkcji int yyparse(). Do uruchomienia parsera zawartego w tej funkcji potrzebujemy dwóch innych funkcji: *main*() i *yylex*().
- Funkcja *main*() wywołuje funkcję *yyparse*().
- Parser wywołuje funkcję *yylex*() w celu pobrania tokena z wejścia.
- YACC definiuje nazwy tokenów w parserze jako nazwy preprocesora C w pliku y.tab.h, więc *yylex*() może je użyć.
- Gdy *yylex*() znajdzie token, zwraca do parsera jego numeryczną wartość, umieszczając ją w zmiennej *yyval*.

Na rysunku 11.4 przedstawiony jest schemat organizacji działania kompilatora zbudowanego za pomocą narzędzi LEX i YACC.



Rys. 11.4. Schemat organizacji działania YACC

Tworzenie pliku specyfikacji. Każdy plik ze specyfikacją dla programu YACC powinien składać się z trzech sekcji. Pierwsza sekcja to sekcja definicji, gdzie umieszczamy definicje i deklaracje zmiennych, stałych, deklaracje stanów oraz makra preprocesora. Sekcja definicji może zawierać fragment kodu, który zostanie uwzględniony przez analizator składniowy. Kod ten musi być odpowiednio „opakowany”: otwarcie fragmentu powinno być poprzedzone znacznikiem %{, natomiast jej zamknięcie – znacznikiem %};

Przykład budowy sekcji definicji:

```
%{
#include <iostream.h>
int zmienna;
int zmienna_druga=1;
%}
```

Druga sekcja to sekcja przetwarzania. W sekcji przetwarzania umieszczamy wszelkie reguły przetwarzania, zgodnie z którymi wygenerowany będzie analizator. Budowa reguły przetwarzania opiera się na dwóch zasadniczych częściach – na produkcji i operacji. Produkcja jest zapisana w notacji programu YACC. Strzałka w produkcji jest zastąpiona znakiem dwukropka (:). Kolejne ciała produkcji, których lewa strona jest taka sama, oddzielamy znakiem pionowej kreski (|). Operacja jest blokiem instrukcji języka C.

Jeśli w gramatyce mamy produkcję: $S \Rightarrow T + T$, to reguła produkcji może być taka:

```
 S  :   T  '+'  T          {printf("liczba + liczba");}
    ;
```

Rozważmy gramatykę, w której zbiór produkcji jest następujący:

```
    E -> E + T;
    E -> T;
    T -> T * F;
    T -> F;
    F -> ( E );
    F -> num;
```

Produkcje w notacji YACC:

```
%token num
%%
E : E '+' T
  | T
  ;
T : T '*' F
  | F
  ;
F : '(' E ')'
  | num
  ;
```

Niech G będzie gramatyką z produkcjami $T \Rightarrow (T)$ i $T \Rightarrow \varepsilon$. Wtedy produkcję możemy zapisać w notacji YACC w sposób następujący:

```
%%
T  :  '(' T ')'
   |
   ;
```

Akcja semantyczna YACC jest sekwencją instrukcji w C. Symbol $$ odwołuje się do wartości atrybutu skojarzonej z nieterminalem po lewej stronie. Symbol $\$i$ odwołuje się do wartości skojarzonej z i-tym symbolem gramatyki po lewej stronie. Akcja semantyczna wywoływana jest zawsze, gdy redukcja odbywa się według związanej z nią produkcji. Rozważmy dwie produkcje: $E \Rightarrow E + T|T$ z notacją YACC:

```
wyr : wyr '+' term  {$$ = $1 + $3;}
    | term
    ;
```

W poniższym fragmencie kodu jest pokazany przykład prostego kalkulatora zapisanego za pomocą produkcji dla narzędzia YACC:

```
%{
#include< ctype.h>
%}
%token DIGIT
%%

line    :expr '\n'          {printf("%d\n",$1);}
        ;

expr    : expr '+' term     {$$=$1+$3}
        | expr '-' term     {$$=$1-$3}
        | term
        ;

term    : term '*' factor    {$$=$1*$3}
        | term '/' factor    {$$=$1/$3}
        | factor
        ;

factor  : '(' expr ')'      {$$=$2}
        | DIGIT
        ;
%%

yylex()
{
    int c;
    c=getchar();
    if isdigit(c)
    {
        yylval=c-'0';
        return DIGIT;
    }
    return c;
}
```

Konflikty. W celu rozwiązania konfliktu YACC stosuje dwie reguły:

1. Konflikt *reduce* / *reduce* jest rozwiązywany przez wybór produkcji, która została umiejscowiona jako pierwsza w specyfikacji YACC.
2. Domyślnie konflikt *shift* / *reduce* zostaje zawsze rozwiązany na rzecz *shift*.

Ponieważ ta ostatnia zasada nie zawsze może być właściwa, YACC zawiera ogólny mechanizm rozwiązywania konfliktów *shift* / *reduce*. W deklaracjach możemy przypisać pierwszeństwo i łączność do terminali. Deklaracja $\%left$ ' + ' ' − ' powoduje, że operatory „+" i „-" będą miały takie same pierwszeństwo i będą łączne lewostronnie.

Możemy zadeklarować operator łączny prawostronnie jako: *%right* ' − '. Pierwszeństwo terminali i produkcji określa kolejność, w której pojawiają się one w części deklaracyjnej. Jeśli YACC ma wybrać między przesunięciem symbolu a a redukcją na podstawie produkcji $A \Rightarrow \alpha$, to wybiera redukcję, jeśli pierwszeństwo produkcji jest większe od pierwszeństwa symbolu a; w innym wypadku wybiera przesunięcie.

Prosty kalkulator. Jego program zapisany jest w dwóch plikach:
calc.l – zawiera specyfikację dla narzędzia Lex,
calc.y – zawiera specyfikację dla narzędzia YACC oraz wywołanie funkcji *yylex*.

Współpraca między LEX i YACC. Przykładowy plik scanner.l przedstawiono poniżej:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{id}        { return ID; }
```

Poniżej znajduje się przykładowy plik parser.y.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token CHAR, FLOAT, ID, INT
%%
```

W kolejnym kodzie programu przedstawiony jest plik dla YACC:

```
%{
#include <stdio.h>
int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS     /*supplies precedence for unary minus */
%%               /* beginning of rules section */
list:            /*empty */
    | list stat '\n'
    | list error '\n' { yyerrok;}
    ;

stat: expr           {printf("%d\n",$1);}
    | LETTER '=' expr {
            regs[$1] = $3;
        }
    ;

expr: '(' expr ')' {
            $$ = $2;
        }
    | expr '*' expr {
            $$ = $1 * $3;
        }
    | expr '/' expr {
            $$ = $1 / $3;
```

```
        }
    | expr '%' expr {
            $$ = $1 % $3;
        }
    | expr '+' expr {
            $$ = $1 + $3;
        }
    | expr '-' expr {
            $$ = $1 - $3;
        }
    | expr '&' expr {
            $$ = $1 & $3;
        }
    | expr '|' expr {
            $$ = $1 | $3;
        }
    | '-' expr %prec UMINUS {
            $$ = -$2;
        }
    | LETTER {
            $$ = regs[$1];
        }
    | number
    ;

number:  DIGIT {
            $$ = $1;
            base = ($1==0) ? 8 : 10;
        }
      | number DIGIT {
            $$ = base * $1 + $2;
        }
      ;

%%
main()
{
     return(yyparse());
}

yyerror(char *s)
{
    fprintf(stderr, "%s\n",s);
}

yywrap()
{
    return 1;
}
```

Poniżej zaprezentowano plik dla narzędzia Lex:

```
%{
#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z] {
        c = yytext[0];
        yylval = c - 'a';
        return LETTER;
      }
[0-9] {
        c = yytext[0];
        yylval = c - '0';
        return DIGIT;
      }
[^a-z0-9\b]{
        c = yytext[0];
        return c;
      }
```

Kompilacja i wykonanie:

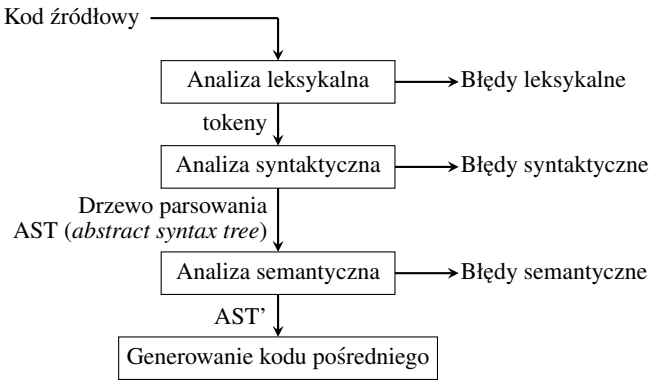
- bison -d –y calc.y – utworzenie plików y.tab.c i y.tab.h
- flex calc.l – utworzenie pliku lex.yy.c
- gcc -g lex.yy.c y.tab.c -o calc – utworzenie pliku wykonywalnego
- ./calc – uruchomienie kalkulatora

12. Analiza semantyczna

Przed bardziej szczegółowym omówieniem analizy semantycznej należy przypomnieć o zadaniach poszczególnych faz analizy; są to:

- analiza leksykalna, która wykrywa nielegalne tokeny, na przykład: main$();
- analiza syntaktyczna, która sprawdza poprawność syntaktyczną programu, na przykład: brak średnika;
- analiza semantyczna, która jako ostatnia faza analizy wykrywa wszystkie pozostałe błędy.

Na rysunku 12.1 pokazany jest schemat działania kompilatora, z uwzględnieniem analizatora semantycznego.



Rys. 12.1. Miejsce analizy semantycznej

Spróbujmy odpowiedzieć na pytanie: Co jest nieprawidłowego w tym kodzie? (brak błędów syntaktycznych):

```
1  foo(int a, char * s){...}
2  int bar() {
3      int f[3];
4      int i, j, k;
5      char *p;
6      float k;
7      foo(f[6], 10, j);
8      break;
9      i->val = 5;
10     j = i + k;
11     printf("%s,%s.\n",p,q);
12     goto label23
13 }
```

W powyższym kodzie błąd powoduje konstrukcja f[6] w linii 7.

Cele analizy semantycznej

1. Kompilator musi zrobić coś więcej poza rozpoznaniem, czy zdanie należy do języka.
2. Znajdowanie pozostałych błędów, które sprawiają, że program jest niepoprawny:
   - niezdeklarowane zmienne, typy,
   - błędy, które mogą zostać wykryte statycznie.
3. Tworzenie danych przydatnych dla późniejszych faz translacji:
   - typy wszystkich wyrażeń,
   - układ danych.

Terminologia:

- kontrola statyczna – wykonana przez kompilator,
- kontrola dynamiczna – realizowana w czasie wykonywania programu.

Rodzaje kontroli

1. Kontrola wyjątkowości:
   - niektóre nazwy muszą być unikatowe,
   - wiele języków wymaga deklaracji zmiennych.
2. Przepływ kontroli sterowania:
   - dopasowanie operatorów sterowania do dozwolonych struktur,
   - przykład: operator *break* zastosowany na zewnątrz konstrukcji *for* / *switch*.
3. Kontrola typów – kontrola zgodności operatorów i operandów.
4. Kontrola logiki programu – program jest składniowo i semantycznie poprawny, ale produkuje zły wynik.

Przykłady błędów na etapie analizy semantycznej:

- niezdefiniowany identyfikator,
- wielokrotnie zadeklarowany identyfikator,
- zmienna iteracyjna pętli poza granicami,
- błędna liczba argumentów funkcji,
- niezgodne typy operandów operatora,
- operator *break* jest na zewnątrz instrukcji *switch* / *for*,
- brak etykiety w instrukcji *goto*.

Kontrola programu – zadania:

- wykrywanie błędów: f[6] w analizowanym wyżej przykładzie powoduje błąd,
- zgłoszenie błędów do programisty,
- wsparcie programisty w celu zweryfikowania jego zamiaru.

W czym ta kontrola jest pomocna?

- przydziela odpowiednią ilość miejsca dla zmiennych,
- wybiera odpowiednie operatory na podstawie operandów,
- wybiera właściwe struktury kontrolne operatorów.

Czy możemy wykryć wszystkie błędy?

```
1 void main()
2 {
3     int i=21, j=42;
4     printf("Hello World\n");
5     printf("Hello World, N=%d\n");
6     printf("Hello World\n", i, j);
7     printf("Hello World, N=%d\n");
8     printf("Hello World, N=%d\n");
9 }
```

Możliwe jest zidentyfikowanie błędów w powyższym kodzie, jeżeli jest określona semantyka analizowanej funkcji.

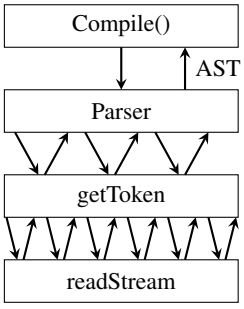
Kontrola typów i generowanie kodu. Można kontrolować typy i wygenerować kod w ramach akcji semantycznych:

```
expr : expr PLUS expr {
    if ($1.type == $3.type &&
        ($1.type == IntType || $1.type == RealType))

        $$.type = $1.type
    else
        error("+ applied on wrong type!");
    GenerateAdd($1, $3, $$);
}
```

Problemy:
- akcje mogą być trudne do odczytania,
- kompilator musi przeanalizować cały program w celu znalezienia błędów.

Na rysunku 12.2 pokazany jest alternatywne podejście do budowy kompilatora, a poniżej opisano to za pomocą pseudokodu:

```
void Compile() {
    AST tree = Parser(program);
    if (TypeCheck(tree)){
        IR ir = GenIntermedCode(tree);
        EmitCode(ir);
    }
}
```



Rys. 12.2. Alternatywne podejście do budowy kompilatora

Typowe błędy semantyczne:
- wielokrotne deklaracje – zmienna musi być zadeklarowana (w tym samym zakresie) co najwyżej raz;
- zmienna niezdeklarowana – zmienna nie może być stosowana bez jej deklaracji;
- niezgodność typów – typ zmiennej po lewej stronie instrukcji przypisania musi pasować do typu wyrażenia prawej strony;
- błędne argumenty – funkcja musi być wywołana z poprawnymi liczbą i typami argumentów.

Prosty analizator semantyczny pracuje w dwóch fazach – przechodzi przez drzewo parsowania utworzone przez parser. Dla każdego zakresu w programie:

1. Przetwarza deklaracje i instrukcje
   - dodaje nowe wpisy do tablicy symboli i zgłasza zmienne, które są wielokrotnie deklarowane;
   - znajduje niezdeklarowane zmienne;
   - aktualizuje węzły odpowiadające nazwom w programie, dodając ich wejścia w tablicy symboli;
2. Przetwarza deklaracje i instrukcje jeszcze raz
   - wykorzystując informacje w tablicy symboli, określa typ każdego wyrażenia oraz wyszukuje błędy typów.

Zakres zmiennych. W większości języków ta sama nazwa może być użyta wielokrotnie, jeśli jej deklaracje występują w różnych zakresach.

W języku Java można używać tej samej nazwy dla:
- klasy,
- pola klasy,
- metody klasy,
- lokalnej zmiennej metody.

```
class Test
{

    int Test;

    void Test( )
    {
        double Test;
    }

    void AnotherTest( )
    {
        double Test;
    }
}
```

W językach Java i C++ można użyć tej samej nazwy dla więcej niż jednej metody, pod warunkiem, że liczba i / lub typy parametrów są unikatowe:

```
int add(int a, int b);
float add(float a, float b);
```

Zasady określania zakresu języka:
- wskazanie, które deklaracje obiektu, mającego nazwę, odpowiadają każdemu użyciu obiektu;
- określenie zakresu użycia obiektów do ich deklaracji za pomocą mapowania.

Języki C++ i Java używają zakresów statycznych (*static scoping*) – mapowanie odbywa sie w czasie kompilacji. C++ wykorzystuje regułę „najściślej zagnieżdżone”:
- zastosowanie zmiennej *x* odpowiada deklaracji w zakresie najściślej otaczającego bloku,
- deklaracja zmiennej poprzedza jej użycie.

Zasięgi nazw (*scope levels*). Każda funkcja ma dwa lub więcej zakresów:

1. Jeden zakres dla ciała funkcji
   - czasami parametry mają osobne zakresy!
   - (nie w języku C)

```
void f( int k )
{
    // k is a parameter
    int k = 0; // also a local variable
    while (k)
    {
        int k = 1; // another local var, in a loop
    }
}
```

2. Dodatkowe zakresy funkcji
   - dla pętli,
   - dla bloku zagnieżdżonego.

Punkt kontrolny 1. Dopasuj każde użycie nazwy do deklaracji nazwy lub znajdź użycie, w przypadku którego brakuje deklaracji:

```
int k=10, x=20;
void foo(int k) {
    int a = x; int x = k; int b = x;
    while (...) {
        int x;
        if (x == k) {
            int k, y;
            k = y = x;
        }
        if (x == k) {
            int x = y;
        }
    }
}
```

Nie wszystkie języki używają zakresu statycznego. Lisp, APL, Snobol używają zakresu dynamicznego (*dynamic scoping*). Zakres dynamiczny oznacza użycie zmiennej, która nie ma jednej stałej deklaracji; jej deklaracja odpowiada deklaracji w ostatnim (względem czasu) wywołaniu wciąż aktywnej funkcji. Na przykład rozważmy kod poniżej:

```
int i = 1;
void func() {
    cout << i << endl;//jeśli C++ zastosował zakres dynamiczny,
        //to zostanie wyświetlona wartość 2, nie 1
}
int main () {
    int i = 2;
    func();
    return 0;
}
```

Punkt kontrolny 2. Zakładając, że dynamiczne zakresy są dozwolone, wskaż, co jest wyświetlane według następującego programu:

```
void main() { int x = 0; f1(); g(); f2(); }
void f1() { int x = 10; g(); }
void f2() { int x = 20; f1(); g(); }
void g() { print(x); }
```

Należy określić sposób, aby śledzić wszystkie typy identyfikatorów w każdym zakresie:

```
{
    int i, n = ...;
    for (i=0; i < n; i++)//i->int, n->int
    boolean b= ...//i->int, n->int, b->boolean
}
```

Tablice symboliczne. Cel: śledzenie nazw zdeklarowanych w programie poprzez wpisy w tablicy symboli:
- typ nazwy(*variable*, *class*, *field*, *method*, ...);
- typ (*int*, *float*, ...);
- poziom zagnieżdżenia;
- adres w pamięci.

Funkcje:
- typ wyszukiwania(np. łańcuch, *string*);

– *void add (String id, Type binding)*;
– wiązania: pary nazwa – typ $a \Rightarrow string, b \Rightarrow int$.

Środowisko reprezentuje zbiór odwzorowań w tablicy symboli:

```
function f(a:int, b:int, c:int) = //σ0
(
    print_int(a+c);
    let var j := a+b          //σ1 = σ0 + j ⇒ int
    var a := "hello"          //σ2 = σ1 + a ⇒ string
    in print(a); print_int(j)
    end;
    print_int(b)
)
```

Zastosowanie tablic symboli:

```
int x;
char y;
void p(void)
{
    double x;
    ...
    {
        int y[10];
        ...
    }
    ...
}
void q(void)
{
    int y;
    ...
    ...
}
main()
{
    char x;
    ...
}
```

W poniższych tabelach (12.1, 12.2, 12.3, 12.4, 12.5, 12.6) przedstawiono stan tablic symboli w poszczególnych liniach programu, oznaczonych za pomocą wielokropka (znaczenie pól w tablicach symboli: *name* – nazwa symbolu, np. zmiennej lub funkcji; *bindings* – wiązania).

Implementacja tablicy symboli. Na rysunku 12.3 pokazana jest zasada działania tablicy symboli. Do implementacji potrzebne są dwie struktury – tablica haszująca, stos zakresów:

Symbol = foo

Hash(foo) = i

Zakres wejścia / wyjścia. Potrzebujemy również stosu do śledzenia „poziomu zagnieżdżenia” w czasie przechodzenia przez drzewo.

Zmienne i typy. Często kompilatory tworzą osobne tablice symboli dla typów zmiennych / funkcji. Czym jest typ? To pojęcie różni się w różnych językach.

Konsensus:
– zbiór wartości,
– zbiór operatorów dozwolonych na tych wartościach.

Tabela 12.1. Stan w linii 6

Name	Bindings	Bindings
x	**double** local to **p**	**int** global
y	**char** global	
p	**void function**	

Tabela 12.2. Stan w linii 9

Name	Bindings	Bindings
x	**double** local to **p**	**int** global
y	**int array** local to nested block in p	**char** global
p	**void function**	

Tabela 12.3. Stan w linii 11 (symbolizuje koniec funkcji w linii 12)

Name	Bindings
x	**int** global
y	**char** global
p	**void function**

Tabela 12.4. Stan w linii 16

Name	Bindings	Bindings
x	**int** global	
y	**int** local to **q**	**char** global
p	**void function**	
q	**void function**	

Tabela 12.5. Stan w linii 17 (symbolizuje koniec funkcji w linii 18)

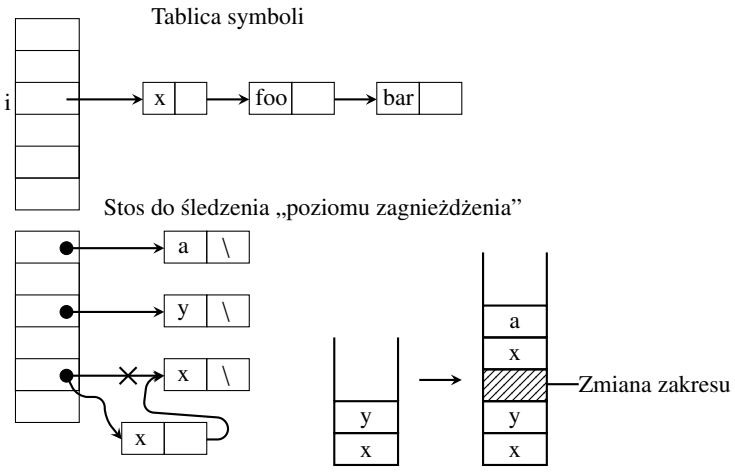
Name	Bindings
x	**int** global
y	**char** global
p	**void function**
q	**void function**

Tabela 12.6. Stan w linii 22

Name	Bindings	Bindings
x	**char** local to **main**	**int** global
y	**char** global	
p	**void function**	
q	**void function**	
main	**int function**	

Niektóre operatory są legalne dla każdego typu. Nie jest sensowne dodawanie wskaźnika na funkcję i liczby całkowitej w C. Sensowne jest to, aby dodać dwie liczby całkowite. Ale obydwa dodawania mają taką samą realizację w asemblerze!

System typów języka. Określa, które operacje są ważne w zależności od typów operandów. Celem kontroli typu jest zapewnienie, że operacje są stosowane na dozwolonych typach operandów. Systemy typów określają zwięzłą formalizację reguł semantycznych sprawdzających poprawność typów dla operatorów. Rozważmy instrukcję w asemblerze:



Rys. 12.3. Implementacja tablicy symboli

```
addi $r1, $r2, $r3
```

Jakie są rodzaje typów dla $\$r1, \$r2, \$r3$? Są cztery rodzaje typów:

- statyczne – wszystkie lub prawie wszystkie sposoby sprawdzania typów są wykonywane w czasie kompilacji;
- dynamiczne – prawie wszystkie sposoby sprawdzania typów odbywają się w czasie przetwarzania programu, np. Perl, Ruby;
- model mieszany – Java;
- brak typu – brak kontroli typów (kod maszynowy).

Kontrola typów i wnioskowanie o typie. Kontrola typów jest to proces sprawdzania typów. Biorąc pod uwagę operator i operandy jakiegoś typu, określa się, czy ten operator jest dozwolony dla danych operandów. Wnioskowanie o typie jest to proces wnioskowania, jaki to jest typ. Biorąc pod uwagę typy poszczególnych operandów, ustala się:

- znaczenie operatora,
- typ operatora.

W przypadku gdy nie ma deklaracji zmiennej określa się jej typ na podstawie wywnioskowania, w jaki sposób zmienna jest używana.

Zasady wnioskowania

- Czy język ma system typów? Języki mogą nie mieć systemu typów(np. Asembler).
- Kiedy są sprawdzane typy? Typy statyczne sprawdzane są w czasie kompilacji, typy dynamiczne – w czasie wykonywania programu.
- Jak ściśle egzekwowane są reguły kontroli? Typy silne – brak wyjątków, typy słabe – ze ściśle określonymi wyjątkami.

Równoważność typów. Kiedy dwa typy są równoważne? Co oznacza równoważność? Kiedy można zastąpić jeden typ innym typem?

Składowe systemu typów:

- typy wbudowane,
- reguły tworzenia nowych typów – gdzie informacja o typach będzie przechowywana,
- reguły określania typów równoważnych,
- zasady wnioskowania o typie wyrażeń.

Typy wbudowane:

1) całkowity – zwykłe operatory – arytmetyka standardowa;
2) zmiennoprzecinkowy – zwykłe operatory – arytmetyka standardowa;
3) znakowy
   - zbiór znaków zwykle jest uporządkowany w sposób leksykograficzny;
   - zwykłe operatory: porównanie leksykograficzne;
4) boolowski – zwykłe operatory: *not*, *and*, *or*, *xor*.

Konstruktory typów

Tablice:

– tablica (I,T) – oznacza typ tablicy z elementami typu T i zbiorem indeksów I;
– tablice wielowymiarowe – tablice, gdzie T jest również tablicą;
– operacje – dostęp do elementu, przypisanie wartości elementom tablicy.

Łańcuchy:

– łańcuchy bitów, łańcuchy znaków;
– operacje: łączenie, porównanie leksykograficzne.

Rekordy (struktury) – grupy wielu obiektów różnych typów, w których elementy mają konkretne nazwy.

Wskaźniki:

– adresy;
– operacje: arytmetyczne, dereferencji, referencji.

Typy funkcji – funkcja $intadd(real, int)$ ma typ $realxint \Rightarrow int$.

Równoważność typów. Typy są równoważne tylko wtedy, gdy mają taką samą nazwę.

Równoważność strukturalna. Typy są równoważne tylko wtedy, gdy mają taką samą strukturę.

Przykład: Język C wykorzystuje równoważność strukturalną dla struktur i równoważność nazw dla tablic i wskaźników.

Wymuszenie typów: Jeśli x ma typ $float$, czy przypisanie $x = 3$ jest dozwolone?

– nie jest dozwolone,
– dozwolone i niejawne, 3 jest konwertowane na typ *float*,
– dozwolone, ale wymaga jawnego konwertowania przez programistę wartości 3 na typ *float*.

Jakie konwertowanie jest dozwolone?

– *float* na *int* ?
– *int* na *float* ?
– czy wielokrotne konwertowanie jest dozwolone?

Podsumowanie: Kompilator musi zrobić coś więcej oprócz rozpoznania, czy zdanie należy do języka:

– znajduje niezdeklarowane zmienne i typy;
– zwraca błędy typów, które mogą zostać wykryte statycznie;
– przechowuje informacje przydatne dla późniejszych faz kompilacji;
– określa typy wszystkich wyrażeń.

13. Prosty translator

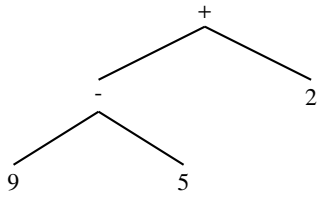
Budowa translatora. Schemat translacji sterowanej składnią często stanowi specyfikację translatora. Schemat na poniższym kodzie zostanie użyty jako definicja translacji, która przekształca wyrażenia arytmetyczne na odwrotną notację polską.

Translator dla prostych wyrażeń (przy gramatyce lewostronnie rekurencyjnej) jest opisany za pomocą następującej gramatyki:

```
1 expr -> expr + term { print('+') }
2         | expr - term { print('-') }
3         | term
4
5 term -> 0       { print ('0') }
6         |  1        { print('1') }
7         ...
8         |  9        { print ('9') }
```

Zastosowanie analizy zstępującej. Gramatyka jest lewostronnie rekurencyjna, więc „parser przewidujący" jej nie obsługuje. Mamy konflikt – z jednej strony potrzebujemy gramatyki, która ułatwia translację, z drugiej strony musimy mieć zupełnie inną gramatykę, która ułatwi parsowanie. Rozwiązanie polega na tym, że rozpoczynamy od gramatyki dla łatwej translacji i starannie przekształcamy ją w celu ułatwienia parsowania. Eliminując rekurencję lewostronną, możemy uzyskać gramatykę odpowiednią do zastosowania parsera przewidującego.

Składnia abstrakcyjna i konkretna. W abstrakcyjnym drzewie syntaktycznym wyrażenia każdemu węzłowi wewnętrznemu odpowiada operator; dzieci węzła reprezentują argumenty operatora. Bardziej ogólnie: każda konstrukcja języka programowania może być obsługiwana przez operator dla tej konstrukcji; operandami tego operatora są semantycznie sensowne elementy tej konstrukcji. Na rysunku 13.1 pokazane jest drzewo abstrakcyjne dla wyrażenia 9 = 5 + 2.



Rys. 13.1. Drzewo abstrakcyjne dla 9 = 5 + 2

Abstrakcyjne drzewo syntaktyczne, lub po prostu drzewo syntaktyczne, przypomina w pewnym stopniu drzewo parsowania. Jednak w drzewie syntaktycznym węzły wewnętrzne reprezentują konstrukcje programistyczne, podczas gdy w drzewie parsowania węzły wewnętrzne reprezentują nieterminale. Wiele symboli nieterminalnych gramatyki reprezentuje konstrukcje programistyczne; inne konstrukcje są to tak zwani pomocnicy. Na przykład symbole *terms* i *factors* (wprowadzone w rozdziale 2) stosowane są do wyprowadzenia wyrażeń arytmetycznych.

W drzewie syntaktycznym symbole „pomocnicy” zazwyczaj nie są potrzebne i są usuwane. Aby podkreślić kontrast, drzewo parsowania jest czasami nazywane konkretnym drzewem syntaktycznym, natomiast odpowiednia gramatyka jest nazywana konkretną składnią języka. Wskazane jest, aby schemat translacji był oparty na gramatyce, której drzewa parsowania byłyby tak blisko drzew syntaktycznych, jak to jest możliwe.

Dostosowanie schematu translacji. Technika eliminacji rekurencji lewostronnej, przedstawiona na rysunku 13.2 (oraz w rozdziale 4), może być zastosowana do produkcji zawierających akcje semantyczne. Rozważmy gramatykę:

```
1  expr -> expr + term     { print('+') }
2        | expr - term     { print('-') }
3        | term
4
5  term -> 0               { print ('0') }
6        |  1              { print('1') }
7        ...
8        |  9              { print ('9') }
```

Zgodnie z techniką eliminacji rekurencji lewostronnej w naszym przykładzie A jest to *expr*; są też dwie produkcje lewostronnie rekurencyjne dla *expr* oraz jedna produkcja, która nie jest rekurencyjna. Technika dokonuje transformacji produkcji $A \Rightarrow A\alpha|A\beta|\gamma$ na produkcje:

$A \Rightarrow \gamma R$

$R \Rightarrow \alpha R \mid \beta R \mid \varepsilon$

Musimy także przekształcić produkcje z akcjami semantycznymi. Akcje semantyczne po prostu są traktowane tak, jak gdyby były one terminalami.

Niech:

$A = expr$

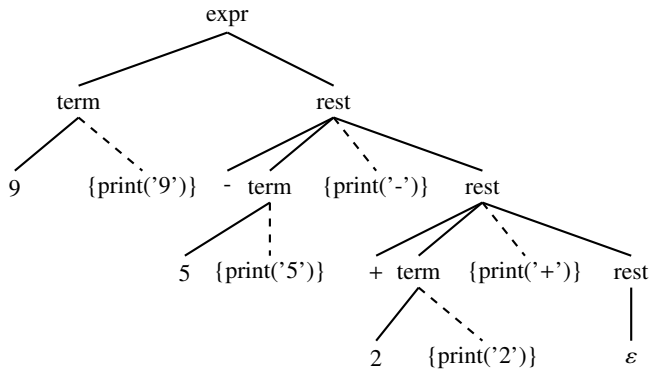
$\alpha = +term\ print('+')$

$\beta = -term\ print('-')$

$\gamma = term$

Wtedy transformacja usuwania rekurencji lewostronnej produkuje następujący schemat:

```
1
2  expr ⇒ term rest
3  rest ⇒ + term           { print('+') }  rest
4       |  - term          { print('-') }  rest
5       |  ε
6  term ⇒ 0                  { print('0') }
7  term ⇒ 1                  { print('1') }
8     ...
9  term ⇒ 9                  { print('9') }
```



Rys. 13.2. Usuwanie rekurencji lewostronnej

Pseudokod dla nieterminali:

```
void expr() {
    term(); rest();
}
void rest() {
    if ( lookahead == '+' ) {
        match('+'); term();
        print('+'); rest();
    }
    else if ( lookahead == '-' ) {
        match('-'); term();
        print('-'); rest();
    }
    else { } //do nothing with the input
}

void term() {
    if ( lookahead is a digit ) {
        t = lookahead; match(lookahead);
        print(t);
    }
    else
    report("syntax error");
}
```

Funkcja *expr* implementuje produkcję dla nieterminala *expr*. Funkcja *rest* implementuje trzy produkcje dla nieterminala *rest*. Stosuje ona pierwszą produkcję, jeśli symbolem bieżącym jest znak: +, drugą produkcję, jeśli symbolem bieżącym jest znak: −, a także produkcję $rest \rightarrow \varepsilon$ we wszystkich innych przypadkach. Dziesięć produkcji dla *term* generuje dziesięć cyfr. Ponieważ każda z tych produkcji generuje i drukuje cyfrę, ten sam kod realizuje je wszystkie. Jeśli test zakończy się pomyślnie, zmienna *t* przechowuje cyfrę reprezentowaną przez symbol bieżący. Należy pamiętać, że funkcja *match* zmienia symbol bieżący, więc cyfra musi być zapisana w celu późniejszego wydrukowania przy przetwarzaniu nieterminali. Jeżeli podczas wykonywania procedury ostatnią czynnością jest wywołanie rekurencyjne tej samej procedury, to mówimy, że jest to rekurencja ogonowa (*tail recursive*). Wywołania rekurencyjne mogą być wtedy zastąpione przez iteracje. W procedurze bez parametrów rekurencję ogonową można wymienić na instrukcję skoku na początek procedury. Dopóki symbolem bieżącym jest znak: + lub: −, procedura *rest* wywołuje procedurę *term*. W przeciwnym razie wykonywanie pętli *while* się kończy. Gdy rekurencja ogonowa jest realizowana przez iteracje, zostaje tylko jedno wywołanie procedury *rest* z wewnątrz procedury *expr*. Dwie procedury mogą być zatem zintegrowane w jedną procedurę poprzez zastąpienie wywołania procedury *rest*() przez jej ciało, jak w poniższym fragmencie kodu.

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

W poniższym kodzie przedstawiona została klasa *Parser* w języku Java:

```
import java.io.*;
class Parser {
    static int lookahead;
    public Parser() throws IOException {
        lookahead = System.in.read();
    }
    void expr() {
        term();
        while ( true ) {
            if ( lookahead == '+' ) {
                match('+'); term();
                System.out.write('+');
                continue;
            }
            else if (lookahead == '-') {
                match('-'); term();
                System.out.write('-');
                continue;
            }
            else
                return;
        }
    }
    void term() throws IOException {
        if (Character.isDigit((char)lookahead)){
            System.out.write((char)lookahead);
            match(lookahead);
        }
        else
            throw new Error("syntax error");
    }
    void match(int t) throws IOException {
        if ( lookahead == t )
            lookahead = System.in.read();
        else
            throw new Error("syntax error");
    }
}
```

Klasa *Parser* zawiera zmienną *lookahead* i funkcje *Parser*, *expr*, *term*, i *match*. Funkcja *Parser* o takiej samej nazwie jak klasa, jest konstruktorem. Jest ona wywoływana automatycznie, gdy obiekt klasy jest tworzony. Konstruktor *Parser* inicjalizuje zmienną *lookahead* przez wczytanie tokena. Funkcja *expr* realizuje nieterminale *expr* i *rest*, *expr* wywołuje funkcje *term*, a następnie napotyka pętlę *while*, która sprawdza, czy *lookahead* pasuje do: + lub do: −. Funkcja *term* korzysta z funkcji *isDigit* należącej do klasy *Character* języka Java, aby sprawdzić, czy symbol *lookahead* jest cyfrą. Konstrukcja (*char*)*lookahead* konwertuje *lookahead* na znak, ponieważ *lookahead* jest zadeklarowana jako liczba całkowita. Funkcja *match* sprawdza terminale, odczytuje następny terminal, jeśli symbol *lookahead* jest dopasowany do *t*; inaczej sygnalizuje błąd, wykonując *throw new Error*(”*syntax error*”);.

Program w języku C:

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int lookahead;

void error()
{
    printf("syntax error\n");
    exit(EXIT_FAILURE);
}
void match(int t)
{
    if (lookahead == t)
        lookahead = getchar();
    else
        error();
}
void term ()
{
    if (isdigit(lookahead))
    {
        putchar(lookahead);
        match(lookahead);
    }
    else
        error();
}
void rest()
{
    if (lookahead == '+') {
        match('+'); term(); putchar('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-'); rest();
    }
    else
        ;
}
void expr()
{
      term();
      rest();
}

int main(void)
{
    lookahead = getchar();
    expr();
    putchar('\n');
    return EXIT_SUCCESS;
}
```

Kolejność wołania funkcji określa lewa strona produkcji:

```
expr ⇒ term rest

rest ⇒  + term                { print('+') }  rest
      |  - term             { print('-') }  rest
      |  ε

term ⇒ 0                    { print('0') }

term ⇒ 1                    { print('1') }

term ⇒ 2                    { print('2') }
   ...
   ...
term ⇒ 9                    { print('9') }
```

Zastosowanie analizy wstępującej z wykorzystaniem narzędzi Lex i Yacc
Plik LEX:

```
%{
#include <stdlib.h>
#include "y.tab.h"
void yyerror(char *);
%}
VAR [a-z]

%%
{VAR}
{
    yylval.value[0]=*yytext;
    yylval.value[1]='\0';
    return VAR;
}

[-+*/"^"\n] return *yytext;
[ \t] ;
. yyerror("Invalid Char");
%%

int yywrap(void)
{
    return 1;
}
```

W powyższym opisie ciąg znaków $[a-z] = a|b|c|...|z$, definiuje terminal *VAR*. Zmienna *yytext* określa wartości atrybutów tokenów za pomocą odpowiedniego pola unii zadeklarowanej w pliku dla YACC.

Chcąc zadeklarować typy symboli, należy najpierw zdefiniować typ atrybutów za pomocą deklaracji *%union*. Deklaracja *%token* definiuje symbole terminalne, natomiast deklaracja *%type* definiuje symbole nieterminalne. W akcji semantycznej dla reguły *exp* : *term* jest kopiowany tekst ze zmiennej $1 do $$; $1 oznacza pierwszy symbol, którym jest *term*. W akcji semantycznej dla *exp* : *exp* + *term* (linie 16–18) fragment kodu w komentarzu oznacza, że symbol znajduje się już w zmiennej $$, następnie za pomocą funkcji *strcat* dopisywany jest tekst z $3 na koniec tekstu w $$. Zmienna $3 zawiera trzeci symbol, którym jest *term*.

Plik YACC:

```
%{
void yyerror(char *);
int yylex(void);
#include<stdio.h>
char prefix[300];
char postfix[300];
char temp[52];
%}
%union   { char value[300]; }
%token <value> VAR
%type <value> exp term fact
%%
program: exp '\n' { printf("%s\n",$1); }
                  ;
exp: term { strcpy($$,$1); }
   | exp '+' term { /*strcat ($$,$1);*/
                    strcat ($$,$3);
                    strcat ($$,"+"); }
   | exp '-' term { /*strcat ($$,$1); */
                    strcat ($$,$3);
                    strcat ($$,"-"); }
   ;
term: fact { strcpy($$,$1); }
    | term '*' fact { strcat ($$,$3);
                      strcat ($$,"*"); }
    | term '/' fact { strcat ($$,$3);
                      strcat ($$,"/"); }
fact: VAR { strcpy($$,$1); }
    |  fact '^' VAR { strcat ($$,$3);
                      strcat($$,"^"); }
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

14. Generowanie kodu maszynowego 1, symulator SPIM

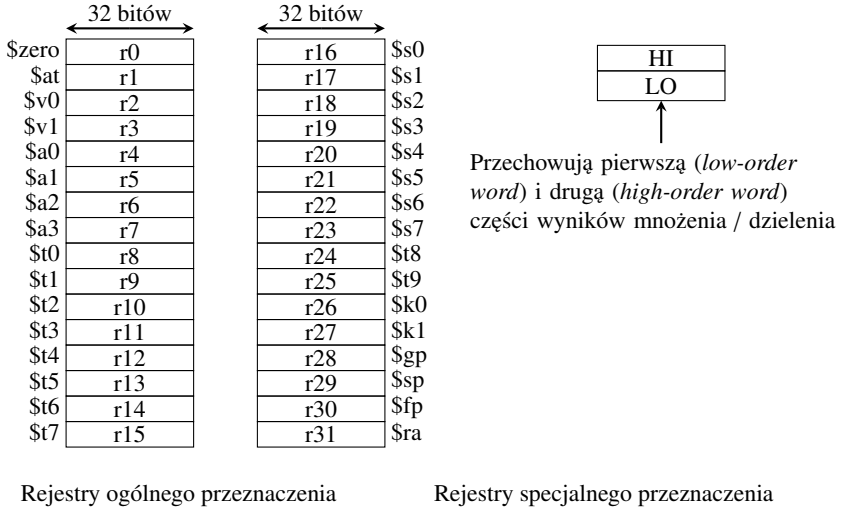
Język Asembler jest symboliczną reprezentacją binarnego kodu komputerowego; jest językiem maszynowym. Asembler jest bardziej czytelny niż kod binarny, ponieważ używa symboli zamiast bitów. Inną jego rolą jest możliwość pisania programu komputerowego. Funkcje systemowe są pisane na podstawie języka Asembler w celu optymalizacji kodu.

Asembler jako program. Assembler jest programem, który przekłada symboliczną wersję kodu na kod binarny.

Symulator SPIM. Jest to symulator, który wykonuje programy napisane w języku Asembler dla procesorów, które implementują architekturę MIPS32. SPIM jest odwróceniem liter nazwy MIPS.

Kod i dokumentacja są dostępne pod adresem: http://spimsimulator.sourceforge.net/. Inne przydatne narzędzie to MIPhpS – Online MIPS Simulator: http://alanhogan.com/asu/simulator.php.

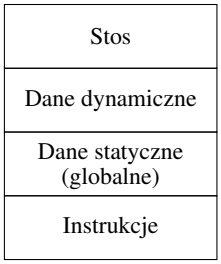
MIPS (Microprocessor without Interlocked Piped Stages). Jest to architektura komputerowa (w szczególności procesor typu RISC) rozwijana przez firmę MIPS Technologies. Istnieje zarówno w wersji 32-, jak i 64-bitowej. Ma 32 rejestry całkowitoliczbowe oraz 32 rejestry zmiennoprzecinkowe. Pierwszy rejestr całkowitoliczbowy jest pseudorejestrem zawierającym zawsze 0 ($zero), co upraszcza wiele operacji. Trzydziesty pierwszy rejestr ($ra) całkowitoliczbowy jest adresem powrotu, przy wywołaniach funkcji. Rejestry MIPS są zaprezentowane na rysunku 14.1. Natomiast na rysunku 14.2 został przedstawiony schemat ogranizacji pamięci.



Rys. 14.1. Rejestry MIPS

Uwagi:

– SPIM wymaga etykiety *main*: w miejscu startu.
– Dane muszą być poprzedzone dyrektywą *.data*.
– Kod wykonywalny musi być poprzedzony dyrektywą *.text*.



Rys. 14.2. MIPS – organizacja pamięci

– Dane i kod mogą być przeplatane.
– Nie można mieć nazw zmiennych, które są takie same jak nazwy rozkazów.

Dyrektywy:

– *.text* – poprzedza kod,
– *.data* – poprzedza dane,
– *.global* – informuje, że symbol jest zmienną globalna,
– *.asciiz* – informuje, że kolejne znaki tworzą ciąg (łańcuch).

Dyrektywy SPIM działają również w innych emulatorach, w tym w MARS. Poniżej przedstawiono przykładowy kod z zastosowaniem dyrektyw:

```
1  .text
2  .globl main
3  main:
4      addi $t0, $zero, 5
5      addi $t1, $zero, 7
6      add $t2, $t0, $t1
7      ...
8      jal swap_proc
9      jr $ra
```

Wielkość bloku danych statycznych pokazanych na rysunku 14.2 jest znana przed kompilacją; czas życia to cały czas wykonywania programu. Pamięć dla danych dynamicznych jest alokowana w czasie realizacji programu. Podobnie jak w przypadku danych dynamicznych wielkość danych na stosie nie jest znana przed kompilacją, np. parametry funkcji są odkładane na stosie, powodując jego powiększenie. Etykieta *.text* zawiera instrukcje programu. Każda dyrektywa w SPIM (również MARS) zaczyna się od kropki.

Dyrektywy SPIM:

```
1  .data
2      .word 5
3      .word 7
4      .byte 25
5      .asciiz "the answer is"
6  .text
7  .globl main
8  main:
9      lw $t0, 0($gp)
10     lw $t1, 4($gp)
11     add $t2, $t0, $t1
12     ...
13     jal swap_proc
14     jr $ra
```

Etykiety:

```
.data
    In1: .word 5
    in2 : .word 7
    C1: .byte 25
    str : .asciiz "the answer is"
.text
.globl main
main:
    lw $t0, in1
    lw $t1, in2
    add $t2, $t0, $t1
    ...
    jal swap_proc
    jr $ra
```

W powyższych przykładach *.globl main* wskazuje, że *main* jest symbolem globalnym, widocznym dla kodu zapisanego w innych plikach. Nazwy *In*1, *in*2, *C*1, *str* pełnią funkcję etykiet. Blok *.data* wskazuje początek danych statycznych.

Poniżej wymieniono instrukcje ładowania i zapisywania:

```
  lw register,addr   - przenosi wartość do rejestru
  li register,num      - przenosi stałą do rejestru
  la register,addr    - przenosi adres do rejestru
  sw register,addr   - zapisuje wartość z rejestru
```

We fragmencie powyżej poszczególne elementy mają następujące znaczenie: litera *l* (instrukcje *lw, li, la*) oznacza *load*, czyli ładowanie wartości (*w – word*), stałej bezpośredniej (*i – immediate*) lub adresu (*a – address*). Litera *s* oznacza *store*, czyli zapis wartości.

Tabela 14.1. Sposoby adresowania

Format	Adres w pamięci
register	zawartość rejestru
imm	bezpośrednia wartość (*immediate*)
imm(register)	bezpośrednia + zawartość rejestru
symbol	adres symbolu
symbol + / – imm	adres symbolu + lub – bezpośrednia
symbol + / – imm(register)	adres symbolu + lub – (bezpośrednia + zawartość rejestru)

Przykłady

Poniższa instrukcja przenosi wartość 5 do rejestru *t*2:

```
li $t2,5
```

Poniższa instrukcja przenosi wartość przechowywaną pod adresem *x* do rejestru *t*3:

```
lw $t3,x
```

Poniższa instrukcja przenosi adres przechowywany pod adresem *x* do rejestru *t*3. *x* oznacza w tym wypadku adres symbolu w tablicy symboli, czyli miejsce w pamięci w danych statycznych.

```
la $t3,x
```

Poniższa instrukcja przenosi wartość, której adresem w pamięci jest zawartość rejestru *t*2, do rejestru *t*0:

```
lw $t0,($t2)
```

Poniższa instrukcja przenosi wartość, której adresem w pamięci jest wartość rejestru $t2$ plus wartość 8, do rejestru $t1$:

```
lw $t1,8($t2)
```

Stosowanie rejestrów. Będziemy korzystać głównie z 8 rejestrów ($\$t0 - \$t7$) do generowania kodu w asemblerze. Dla binarnych operatorów arytmetycznych korzystamy z rejestrów $reg1, reg2, reg3$, jak niżej ($reg1 = reg2\ op\ reg3$):

```
    add reg1,reg2,reg3   (dodawanie)
    sub reg1,reg2,reg3   (odejmowanie)
    mul reg1,reg2,reg3   (mnożenie)
    div reg1,reg2,reg3   (dzielenie)
```

Dla jednoargumentowych operatorów arytmetycznych korzystamy z $reg1, reg2$, jak niżej ($reg1 = op\ reg2$):

```
neg reg1, reg2 ##negowanie wartości
```

Przykład generowania kodu dla instrukcji $a := b * -c + b * -c$
Poniższa instrukcja przenosi wartość b do rejestru $t0$:

```
lw $t0,b
```

Następujący fragment kodu przenosi wartość c do rejestru $t1$:

```
lw $t0,b
lw $t1,c
```

Poniższy fragment neguje wartość $c (c = -c)$; wynik zapisuje do rejestru $t1$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
```

Poniższe instrukcje obliczają wartość $b * -c$; wynik zapisywany jest do rejestru $t1$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
```

Poniższy fragment kodu przenosi wartość b do rejestru $t0$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
```

Następujący fragment kodu przenosi wartość c do rejestru $t2$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
```

Poniższy fragment kodu neguje wartość c, wynik zapisuje do rejestru $t2$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t2
```

Następujący fragment kodu oblicza wartość $b * -c$; wynik zapisuje do rejestru $t0$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
```

W kolejnym fragmencie kodu obliczana jest wartość $b * -c + b * -c$; wynik zapisywany jest do rejestru $t1$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
add $t1,$t0,$t1
```

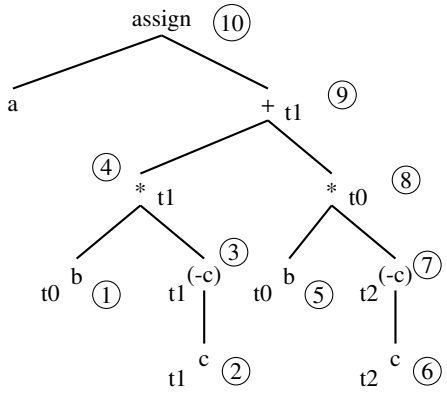
Poniższy kod zapisuje wynik końcowy do rejestru $t1$:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
add $t1,$t0,$t1
sw $t1,a
```

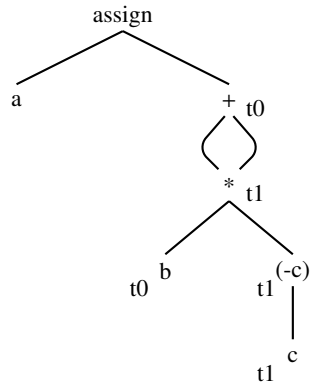
Na rysunku 14.3 oznaczono za pomocą liczb w okręgach poszczególne linie kodu wynikowego (a zarazem kroki parsowania wyrażenia).

Na rysunku 14.4 i w poniższym fragmencie kodu przedstawiono kod zoptymalizowany, w którym nie obliczamy po raz drugi wartości $b * -c$; korzystamy z wcześniej obliczonej wartości:

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1,$t1,$t0
add $t0,$t1,$t1
sw $t0,a
```



Rys. 14.3. Drzewo dla wyrażenia $a = b * -c + b * -c$



Rys. 14.4. Drzewo dla zoptymalizowanego wyrażenia $a = b * -c + b * -c$

Operatory porównania mają następującą strukturę: *temp*1 = *temp*2 *xxx temp*3, gdzie *xxx* oznacza warunek: *sgt*(>), *sge*(>=), *slt*(<), *sle*(<=), *seq*(==), a *temp*1 służy do przechowywania wyniku i wynosi 0 dla *false*; wartość niezerowa oznacza *true*. Znaczenie poszczególnych instrukcji:

- *sgt* – set greater than,
- *sge* – set greater than or equal,
- *slt* – set less than,
- *sle* – set less than or equal,
- *seq* – set equal.

Przykład użycia operatora porównania:

```
    sgt reg1,reg2,reg3
    slt reg1,reg2,reg3
```

Skoki:

1. *b label* – bezwarunkowy skok do etykiety (b – *branch to label*);
2. *bxxx temp*, *label* – warunkowy skok do etykiety, xxx = warunek, np.:
   - eqz(=0) (*branch on equal to zero*),
   - neq(/=) (*branch on not equal*),
   - le(<=) (*branch on less or equal*);
3. *jal label* – skok i zapisanie adresu powrotu (jal – *jump and link*);
4. *jr register* – skok pod adres przechowywany w rejestrze (jr – *jump register*);

Na przykładzie kolejnych fragmentów kodu omówiono przepływ sterowania dla pętli *while*:

```
while x <= 100 do
    x := x + 1
end while
```

Kod powyższej pętli jest zamieniany na następujący fragment kodu wynikowego:

```
1         lw $t0,x
2         li $t1,100
3 L25:    sle $t2,$t0,$t1 ;Set less than or equal
4         beqz $t2,L26
5         addi $t0,$t0,1 ;Addition immediate
6         sw $t0,x
7         b L25
8 L26:
```

W powyższym kodzie linia 4 powoduje skok, jeżeli wartość to fałsz, a linie 5 i 6 są ciałem pętli.

Poniżej podano przykład generowania przez kompilator kodu dla pętli i wyrażeń dla prostej operacji generowania liczb pierwszych.

```
1  print 2  print  blank  #drukuj 2, drukuj spacje
2  for i = 3 to 100
3     divides = 0
4     for j = 2 to i/2
5        if  "reszta z  dzielenia  i  przez j  jest 0" then
6            divides = 1
7     end for
8     if divides = 0 then
9         print i
10        print blank
11   end for
12   exit
```

Na początku generujemy kod dla pętli z linii 2–9. Poniżej znajduje się przetworzona przez kompilator pętla zewnętrzna *for* $i = 3$ *to* 100.

```
1       li $t0, 3          # variable i=3 in t0
2       li $t1,100         # max loop counter in t1
3 l1:   sle $t7,$t0,$t1    # i <= 100
4       beqz $t7, l2
5       ...
6       ...
7       addi $t0,$t0,1     # increment i
8       b l1
9 l2:
```

Poniższy fragment przedstawia pętlę wewnętrzną *for* $j = 2$ *to* $i/2$ (linie 4–7).

```
1      li $t2,2           # j = 2 in t2
2         div $t3,$t0,2   # i/2 in t3
3 l3:   sle $t7,$t2,$t3   # j <= i/2
4       beqz $t7,l4
5       ...
6       ...
7       addi $t2,$t2,1    # increment j
8       b l3
9 l4:
```

Natomiast ostatni fragment dotyczy instrukcji warunkowych (linie 5–6 i 8).

```
1     rem $t7,$t0,$t2     # reszta  i/j
2     bnez $t7,l5         #
3                         #
4     li $t4,1            # divides=1
5  l5:
6     ....
7     bnez $t4,l6         # if divides = 0,
8                            #then print i
9     print i
10 l6:
```

W powyższym kodzie skok (*bnez*) jest wykonywany, jeśli wartość w rejestrze nie jest zerem.

Wywołania systemowe. SPIM zapewnia kilka usług SO: Najbardziej przydatne są operacje I/O: czytania, pisania, otwierania i zamykania plików. Argumenty dla procedury *syscall* są umieszczone w rejestrach $*a*0 – $*a*3. Typ procedury *syscall* jest identyfikowany przez umieszczenie odpowiedniego numeru w rejestrze $*v*0:

- 1 dla *print_int*,
- 4 dla *print_string*,
- 5 dla *read_int*.

Rejestr $*v*0 może przechowywać także adres do zwracania wartości przez wywołanie systemowe.

Poniższe wywołanie odpowiada instukcji *Print*(*i*):

```
1  li $v0,1
2  lw $a0,i
3  syscall
```

W powyższym kodzie liczba 1 ładowana jest do rejestru $*v*0 i określa funkcję *print_int*. Instrukcja *lw* przenosi wartość z adresu *i* do rejestru *a*0; wartość ta jest argumentem funkcji *print_int*.

Poniższe wywołanie odpowiada instukcji *Read*(*i*):

```
1  li $v0,5
2  syscall
3  sw $v0,i
```

W powyższym kodzie liczba 5 ładowana jest do rejestru $*v*0 i określa funkcję *read_int*. Instrukcja *sw* zapisuje w pamięci wartość przechowywaną w $*v*0 pod adresem *i*.

Poniższe wywołanie odpowiada zakończeniu programu (*exit*):

```
1  li $v0,10
2  syscall
```

W powyższym kodzie liczba 10 jest ładowana do rejestru $*v*0 i określa funkcję *exit*, która po wywołaniu *syscall* kończy wykonywanie kodu.

W tabeli 14.2 zostały przedstawione podstawowe funkcje systemowe wraz ze sposobem przekazywania do nich parametrów i pobierania danych wyjściowych.

Poniżej podano jeszcze raz przykład generowania liczb pierwszych, wraz z pełnym kodem wynikowym (w kolejnej ramce).

```
1  print 2  print blank
2  for i = 3 to 100
3     divides = 0
4     for j = 2 to i/2
5        if j divides i evenly then divides = 1
6     end for
7    if divides = 0 then print i print blank
8  end for
9  exit
```

Tabela 14.2. Funkcje systemowe

Funkcja	Kod	Wejście	Wyjście
print_int	$v0 = 1	$a0 = integer to print	prints $a0 to standard output
print_float	$v0 = 2	$f12 = float to print	prints $f12 to standard output
print_double	$v0 = 3	$f12 = double to print	prints $f12 to standard output
print_string	$v0 = 4	$a0 = address of first character	prints a character string to standard output
read_int	$v0 = 5		integer read from standard input placed in $v0
read_float	$v0 = 6		float read from standard input placed in $f0
read_double	$v0 = 7		double read from standard input placed in $f0
read_string	$v0 = 8	$a0 = address to place string, $a1 = max string length	reads standard input into address in $a0
sbrk	$v0 = 9	$a0 = number of bytes required	$v0 = address of allocated memory. Allocates memory from the heap
exit	$v0 = 10		
print_char	$v0 = 11	$a0 = character (low 8 bits)	-
read_char	$v0 = 12		$v0 = character (no line feed) echoed
file_open	$v0 = 13	$a0 = full path (zero terminated string with no line feed), $a1 = flags, $a2 = UNIX octal file mode (0644 for rw-r–r–)	$v0 = file descriptor
file_read	$v0 = 14	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to read in bytes	$v0 = amount of data in buffer from file (-1 = error, 0 = end of file)
file_write	$v0 = 15	$a0 = file descriptor, $a1 = buffer address, $a2 = amount to write in bytes	$v0 = amount of data in buffer to file (-1 = error, 0 = end of file)
file_close	$v0 = 16	$a0 = file descriptor	

Kod wynikowy:

```
.data
blank:  .asciiz " "
.text
    li $v0,1
    li $a0,2
    syscall          # print 2
    li $v0,4
    la $a0,blank   # print blank
    syscall

    li $v0,1
    lw $a0,i
    syscall    # print i

    li $v0,10
    syscall     # exit

.data
    blank:  .asciiz " "
.text
main:
    li $v0,1
    li $a0,2
    syscall
    li $v0,4
    la $a0,blank
```

```
27      syscall
28      li $t0,3     # i in t0
29      li $t1,100   # max in t1
30  l1: sle $t7,$t0,$t1
31      beqz $t7,l2
32      li $t4,0
33      li $t2,2    # jj in t2
34      div $t3,$t0,2 # max in t3
35  l3: sle $t7,$t2,$t3
36      beqz $t7,l4
37      rem $t7,$t0,$t2
38      bnez $t7,l5
39      li $t4,1
40  l5: addi $t2,$t2,1
41      b l3         #end of inner loop
42  l4: bnez $t4,l6
43      li $v0,1
44      move $a0,$t0
45      syscall # print i
46      li $v0,4
47      la $a0,blank
48      syscall
49  l6: addi $t0,$t0,1
50      b l1         #end of outer loop
51  l2: li $v0,10
52      syscall
```

Komentarz do powyższego kodu:

– Dyrektywa *.asciiz*" " zapisuje znaki łańcucha w pamięci.
– Pętla zewnętrzna rozpoczyna się od linii 28.
– W linii 38 jest skok warunkowy do etykiety l6.
– Pętla zewnętrzna kończy się w linii 53.

15. Generowanie kodu maszynowego 2, symulator SPIM

Generowanie kodu. Do wygenerowania kodu w asemblerze potrzebne są:
- deklaracje,
- wyrażenia,
- przepływ sterowania,
- wywołanie procedur.

Przetwarzanie deklaracji polega na sprawdzeniu przez generator kodu kilku informacji:
- czy zmienna lokalna czy globalna;
- jak przydzielić pamięć dla zmiennych;
- jakie podstawowe typy występują: *integer*, *boolean*, ...;
- jakie złożone typy występują: *records*, *arrays*, ...

Poniżej znajduje się fragment opisujący przydział pamięci; jest to kod wygenerowany na podstawie deklaracji.

```
1   .data
2 var_name1:      .word  0
3 var_name2:      .word 29,10
4 var_name3:      .space 40
5 var_name4:      .space 80
```

W powyższym kodzie instrukcje w liniach 2 i 3 oznaczają przydział 4 bajtów do każdego słowa; dodatkowo w linii 3 zaprezentowana została inicjalizacja wartością początkową, a w linii 4 i 5 przydzielone zostały większe obszary pamięci.

Podstawowe dyrektywy SPIM:
- *.data* – poprzedza dane,
- *.ascii* "*str*" – zapisuje *str* w pamięci bez znaku końca wiersza \0;
- *.asciiz* "*str*" – to samo jak wyżej, ale z \0;
- *.byte* $3, 4, 16$ – zapisuje 3 wartości; każda zajmuje jeden bajt;
- *.double* $3.14, 2.72$ – zapisuje 2 wartości zmiennoprzecinkowe z podwójną dokładnością,
- *.float* $3.14, 2.72$ – zapisuje 2 wartości zmiennoprzecinkowe,
- *.word* $3, 4, 16$ – zapisuje 3 wartości; każda zajmuje 32 bity;
- *.space* 100 – rezerwuje 100 bajtów;
- *.text* – zaczyna segment tekstu z instrukcjami.

Przetwarzanie wyrażeń polega na:
- generowaniu poprawnego kodu,
- kontroli typów,
- obliczaniu adresów elementów tablicy,
- obliczaniu wyrażeń warunkowych w konstrukcjach sterowania,
- generowaniu skoków w konstrukcjach sterowania.

Na rysunku 15.1 przedstawione zostało drzewo parsowania i kolejność generowania kodu z poniższego fragmentu kodu obliczającego wyrażenie $a = b + c + d + e$.

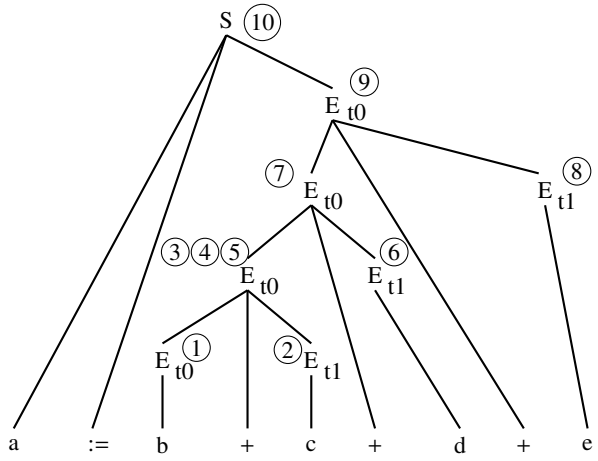
Napisy $t0$ i $t1$ oznaczają użyte do przechowania danej wartości zmienne tymczasowe, natomiast liczby w okręgach oznaczają numer linii z poniższego kodu:

```
1  lw t0,b
2  lw t1,c
3  add $t0,$t0,$t1
4  sw $t0,tmp1
5  lw $t0,tmp1
6  lw t1,d
7  add $t0,$t0,$t1
8  lw $t1,e
9  add $t0,$t0,$t1
10 sw $t0,a
```

Instrukcje z linii 4 i 5:

```
1 sw tmp1,$t0
2 lw $t0,tmp1
```

Są one redundatne, a więc nie ma konieczności ich zapisywania.



Rys. 15.1. Generowanie kodu dla wyrażenia $a = b + c + d + e$

W akcjach semantycznych z tabeli 15.1 *sw* oznacza zapis wartości przechowywanej w rejestrze $3.*reg* w pamięci pod adresem $1; symbol $3 odwołuje się do wartości skojarzonej z trzecim symbolem gramatyki po prawej stronie. W drugiej akcji semantycznej symbol $1 oznacza rejestr, który przechowuje wartość pierwszej nazwy, a symbol $3 oznacza rejestr, który przechowuje wartość drugiej nazwy. Symbol $$ odwołuje się do wartości atrybutu skojarzonej z nieterminalem po lewej stronie. Instukcja *lw* przenosi wartość do rejestru; funkcja *get_register* zwraca wolny rejestr. W trzeciej akcji semantycznej $1 oznacza adres, pod którym w pamięci jest przechowywana wartość zmiennej (id).

Obliczenie adresów elementów tablic. Załóżmy, że b oznacza adres bazowy, od którego zaczyna się obszar pamięci, zarezerwowany do przechowywania elementów tablicy $a[l..h]$; każdy element zajmuje s bajtów. Liczba elementów może być obliczona za pomocą wzoru: $e = h - l + 1$. Rozmiar tablicy: $e * s$. Adres elementu to $a[i]$, przy założeniu, że obszar zaczyna się od adresu $b, l <= i <= h$: $b + (i - l) * s$. Rozmieszczenie pamięci w tablicy jest przedstawione w tabeli 15.2, a w postaci uwzględniającej adresy bezwględne – w tabeli 15.3.

Przykład: a[3...100]; każdy element tablicy jest reprezentowany przez 4 bajty. Liczba elementów: $100 - 3 + 1 = 98$. Rozmiar tablicy: $98 * 4 = 392$. Adres elementu to $a[50]$, przy założeniu, że obszar zaczyna się od adresu 100: $100 + (50 - 3) * 4 = 288$.

Tabela 15.1. Akcje semantyczne dla wyrażeń

Produkcja	Akcja semantyczna
S -> id := E	`{ printf("sw $t%d,%s\n", $3.reg,$1); free_reg($3.reg); //zwalnia rejestr }`
E -> E + E	`{ $$.reg = $1.reg; printf("add $t%d, $t%d, $t%d\n", $$.reg,$1.reg, $3.reg, $3.reg); free_reg($3.reg); //zwalnia rejestr }`
E -> id	`{ $$.reg = get_register(); printf("lw $t%d,%s\n", $$.reg,$1); }`

Tabela 15.2. Elementy tablicy jednowymiarowej

a[l]	a[l+1]	a[l+1]	...	a[h]
b				

Tabela 15.3. Elementy tablicy jednowymiarowej - adresy bezwględne

a[3]	a[4]	a[5]		a[100]
100	104			

Kod w języku C z odwołaniem się do tablicy:

```
1 A[8] = h + A[8];
```

Odpowiadający powyższemu fragmentowi wynikowy kod MIPS, przy spełnieniu następujących założeń:
1) $s3 zawiera adres pierwszego elementu *A* (adres bazowy *b*),
2) $s2 zawiera wartość *h*,
ma postać:

```
1 lw   $t0,32($s3)      # $t0 gets A[8]
2 # i-1 =8, s=4,  (i-1)xs =8 x 4 =32
3 add  $t0,$s2,$t0      # add h
4 sw   $t0,32($s3)      # store value back in A[8]
```

Kod w języku C z odwołaniem się do tablicy przez wartość zmiennej:

```
1 g = h + A [i];
```

Jeżeli spełnione jest założenie, że $s4 zawiera i, to zostanie wygenerowany poniższy wynikowy kod MIPS, odpowiadający fragmentowi kodu z odwołaniem do tablicy poprzez wartość zmiennej:

```
# zapisz wartosc w \$t1
add $t1, $s4, $s4   # $t1 = 2 *i
add $t1, $t1, $t1   # $t1 = 4 *i
# Baza jest przechowywana  w $s3
# Adres  A[i]
add    $t1, $t1, $s3    #$t1=Adres(A[i])
#zapisz A[i]
lw  $t0, 0($t1) # $t0 = A[i]
# dodaj  A[i] do h
add     $s1, $s2, $t0   # $s1 = h + A[i]
```

Tablice dwuwymiarowe

Zapis wierszami i kolumnami dla deklaracji $a[4..6, 3..4]$ pokazano w tabeli 15.4.

Tabela 15.4. Elementy tablicy dwuwymiarowej – zapis wierszami i kolumnami

Adres	Wiersz	Kolumna
b + 0s	a[4,3]	a[4,3]
b + 1s	a[4,4]	a[5,3]
b + 2s	a[5,3]	a[6,3]
b + 3s	a[5,4]	a[4,4]
b + 4s	a[6,3]	a[5,4]
b + 5s	a[6,4]	a[6,4]

Dla tablicy zadeklarowanej jako A[4..7,3..4] otrzymujemy pokazany w tabeli 15.5 zapis wierszami.

Tabela 15.5. Elementy tablicy dwuwymiarowej – zapis wierszami

Adres	Wiersz
b + 0s	a[4,3]
b + 1s	a[4,4]
b + 2s	a[5,3]
b + 3s	a[5,4]
b + 4s	a[6,3]
b + 5s	a[6,4]
b + 6s	a[7,3]
b + 7s	a[7,4]

$a[l_1..h_1, l_2..h_2]$; każdy element jest reprezentowany przez s bajtów.
Liczba elementów: $e = e_1 * e_2$, gdzie $e_1 = (h_1 - l_1 + 1)$ i $e_2 = (h_2 - l_2 + 1)$.
Rozmiar tablicy: $e * s$.
Rozmiar każdego wymiaru: $d_1 = e_2 * d_2$, $e_2 = (h_2 - l_2 + 1)$ $d_2 = s$.
Adres elementu $a[i, j]$ z bazą b, $l_1 <= i <= h_1$, $l_2 <= j <= h_2$: $b + (i - l_1) * d_1 + (j - l_2) * s$ e_2 określa liczbę słów w jednym wierszu.
d_1 określa liczbę bajtów, które zajmuje jeden wiersz.
Iloczyn $(i - l_1) * d_1$ określa liczbę bajtów zajmowanych przez $(i - l_1)$ wierszy.
Iloczyn $(j - l_2) * s$ określa liczbę bajtów , które zajmują $(j - l_2)$ słów.

Przykład:

A[3..100,4..50]; każdy element jest reprezentowany przez 4 bajty.
$98 * 47 = 4606$ elementów,
$4606 * 4 = 18424$ bajtów,
$d_2 = 4, d_1 = 47 * 4 = 188$.

Dla $b = 100$; adres $a[5, 5]$: $100 + (5 - 3) * 188 + (5 - 4) * 4 = 720$

a[3,5] : zapis wierszami; przydział pamięci:

```
.data
a:    .space 60     # 3x5=15 word-size elements * 4
```

Obliczanie adresu:
Adres = $b + (i - l_1) * d_1 + (j - l_2) * s$
$d_1 = e_2 * d_2 = 5 * 4 = 20$
$e_2 = (h_2 - l_2 + 1) = 5$
$d_2 = s = 4$

```
la $t0,a            #baza b w  $t0
lw $t1,x            # x w $t1
mul $t1,$t1,20      # (x - l1) * d1 , d1=20
add $t0,$t0,$t1     # b+ (x- l1)*d1 w $t0
lw $t1,y            # y w $t1
mul $t1,$t1,4       # (j - l2) * s, s=4
add $t0,$t0,$t1     # Adres dla a[x,y]: b + (i - l1) * d1 + (j - l2) * s
lw $t1,($t0)        #t1 zawiera  a[x,y]
```

Tablice 3D: a[4..7,3..4,8..9]
Rozmiar trzeciego wymiaru = s
Rozmiar drugiego wymiaru = $s * 2$
Rozmiar pierwszego wymiaru = $s * 2 * 2$

Tabela 15.6. Elementy tablicy trójwymiarowej

Adres	Odwołanie
b + 0s	a[4,3,8]
b + 1s	a[4,3,9]
b + 2s	a[4,4,8]
b + 3s	a[4,4,9]
b + 4s	a[5,3,8]
b + 5s	a[5,3,9]
b + 6s	a[5,4,8]
b + 7s	a[5,4,9]
b + 8s	a[6,3,8]
b + 9s	a[6,3,9]
b + 10s	a[6,4,8]
b + 11s	a[6,4,9]
b + 12s	a[7,3,8]
b + 13s	a[7,3,9]
b + 14s	a[7,4,8]
b + 15s	a[7,4,9]

$a[l_1..h_1, l_2..h_2, l_3..h_3]$; każdy element zajmuje s bajtów.
Liczba elementów: $e = e_1 * e_2 * e_3$, gdzie: $e_i = (h_i - l_i + 1)$.
Rozmiar tablicy: $e * s$.
Rozmiar poszczególnych wymiarów: $d_1 = e_2 * d_2$, $d_2 = e_3 * d_3$, $d_3 = s$.
Adres elementu $a[i, j, k]$ z bazą b, $l_1 <= i <= h_1$ i $l_2 <= j <= h_2$: $b + (i - l_1) * d_1 + (j - l_2) * d_2 + (k - l_3) * s$.

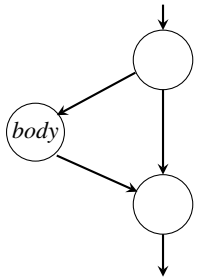
Przykład:
A[3..100,4..50,1..4]; każdy element zajmuje 4 bajty:
$98 * 47 * 4 = 18424$ elementów,
$18424 * 4 = 73696$ bajtów,
$d_3 = 4$, $d_2 = 4 * 4 = 16$, $d_1 = 16 * 47 = 752$.
Dla $b = 100$ adres elementu $a[5, 5, 2]$:
$100 + (5 - 3) * 752 + (5 - 4) * 16 + (2 - 1) * 4 = 1624$.

Przetwarzanie konstrukcji sterowania. Konstrukcje sterowania występujące w językach programowania:
– *if*,
– *while*,
– *repeat*,
– *for*,
– *case*.
Generacja etykiet – wszystkie etykiety muszą być unikatowe.

Instrukcje warunkowe. Na rysunku 15.2 oraz w poniższych fragmentach kodu przedstawiono schemat generowania kodu i przepływ instrukcji dla prostej instrukcji warunkowej.

```
if (y > 0) then begin
...body...
end
```

```
lw $t0,y
li $t1,0
sgt $t2,$t0,$t1  # = 1 if true
beqz $t2,L2

...body...

L2:
```

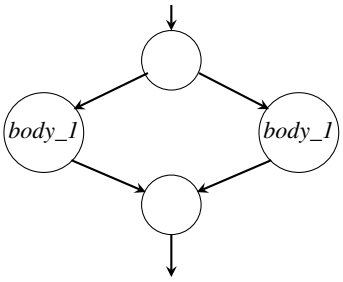


Rys. 15.2. Przepływ danych dla prostej instrukcji warunkowej

Na rysunku 15.3 oraz w poniższych fragmentach kodu przedstawiono schemat generowania kodu i przepływ instrukcji dla instrukcji warunkowej z *else*.

```
if (y > 0) then begin
... body_1 ...
end else
...body_2 ...
end
```

```
lw $t0,y
li $t1,0
sgt $t2,$t0,$t1  # = 1 if true
beqz $t2,L2
...body_1..
b L3
L2:
...body_2 ...
L3:
```

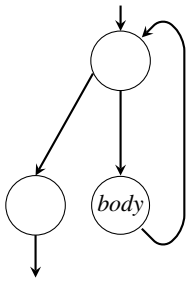


Rys. 15.3. Przepływ danych dla instrukcji warunkowej z *else*

Pętle. Na rysunku 15.4 oraz w poniższych fragmentach kodu przedstawiono schemat generowania kodu i przepływ instrukcji dla pętli.

```
while x < 100 do

 ...body ...

end
```

```
L25:      lw $t0,x
          li $t1,100
          sgt $t2,$t0,$t1
          beqz $t2,L26
          ... body ...
          b L25
L26:
```



Rys. 15.4. Przepływ danych dla instrukcji pętli

Schemat ogólny dla instrukcji warunkowych:

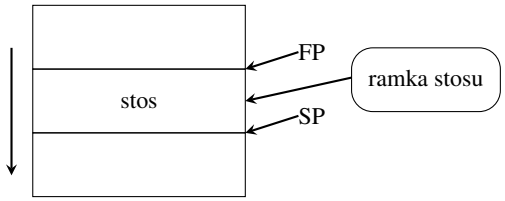
```
if_stmt ->  IF expr THEN
   kod do obliczenia  expr ($2),
   utworz dwie nowe etykiety: L1, L2,
   jesli expr=false ($2=false), to skok do L1,
   cialo if_stmt
   skok do L2
        ELSE
   L1:ciało else_stmt }
        ENDIF
   L2:...
```

Schemat ogólny dla pętli:

```
for_stmt ->  FOR id = start TO stop
{ kod do obliczenia start ($1) i stop ($2),
   utworz 2 etykiety L1, L2,
   utworz kod dla instrukcji id = start,
   L1:kod  do  porownania id i stop ($3),
   jesli ($3)=false, to skok do L2,
   kod dla ciała pętli,
   kod dla inkrementacji  id,
   skok do L1
END FOR
   L2:...
```

Wywołanie procedur. Założenie: jedna funkcja (wywołująca) wywołuje drugą funkcję (wywoływaną). Jakie są czynności realizowane przez pierwszą i drugą funkcję?

Wskaźniki SP i FP – dla każdej funkcji trzeba przydzielić ramkę stosu. FP wskazuje na początek bieżącej aktywacji (pierwsze słowo ramki stosu). SP wskazuje na ostanie słowo ramki stosu. Stos rośnie w kierunku niższych adresów, dlatego chcąc zwiększyć stos, trzeba zastosować operator odejmowania (!). Na rysunku 15.5 przedstawiona jest budowa stosu i umiejscowienie wskaźników SP i FP.



Rys. 15.5. Budowa stosu

Funkcja wywołująca zapisuje argumenty funkcji wywoływanej w standardowych miejscach i wykonuje następujące czynności:

1. Przekazuje argumenty. Zgodnie z konwencją; pierwsze cztery argumenty przekazywane są do rejestrów $\$a0$ – $\$a3$. Wszystkie pozostałe argumenty są odkładane na stos i pojawiają się na początku stosu.
2. Procedura wywoływana może korzystać z następujących rejestrów ($\$a0$ – $\$a3$ i $\$t0$ – $\$t9$). Jeśli funkcja wywołująca zamierza korzystać z tych rejestrów, to musi zapisać zawartość tych rejestrów w pamięci przed wywołaniem.
3. Wykonuje instrukcję *jal*, która przekazuje sterowanie do pierwszej instrukcji funkcji wywołanej i zapisuje adres powrotu w rejestrze $\$ra$.

Przed wykonaniem obliczeń funkcja wywoływana musi wykonać następujące kroki:

1. Przydzielić obszar pamięci (ramkę) poprzez odjęcie wielkości ramki od wskaźnika stosu **FS** (stos zaczyna się od większych adresów).
2. Zapisać zawartość rejestrów funkcji wywoływanej w przydzielonym obszarze pamięci (w ramce). Funkcja wywoływana musi zapisać w ramce dane przechowywane w rejestrach ($\$s0$ – $\$s7$, $\$fp$, $\$ra$) przed korzystaniem z tych rejestrów, ponieważ funkcja wywołująca spodziewa się korzystać z danych w tych rejestrach po zakończeniu wykonania funkcji wywoływanej. Zawartość rejestru $\$fp$ musi być zapisana w pamięci przez każdą procedurę, która przydziela obszar pamięci dla stosu. Natomiast zawartość rejestru $\$ra$ musi być zapisywana tylko wtedy, gdy funkcja wywoływana sama wywołuje inną funkcję. Zawartość pozostałych rejestrów, z których korzysta funkcja wywoływana, musi być także zapisana.
3. Ustawić wskaźnik stosu, dodając rozmiar ramki minus 4 (co zwiększa stos o 4 bajty) do zawartości rejestru $\$sp$ i zapisać wynik w rejestrze $\$fp$.

Zakończenie:

1. Jeśli funkcja wywoływana zwraca wartość, to zapisuje ją w rejestrze $v0.
2. Funkcja wywoływana przywraca zawartość wszystkich rejestrów, które zostały zapisane w momencie wywołania procedury.

3. Funkcja wywoływana zdejmuje ramkę stosu, dodając rozmiar ramki do $sp.
4. Następuje powrót do adresu podanego w rejestrze $ra.

Przykład w C:

```
int main()
{
    x=addthem(a,b);
}
int addthem(int a, int b)
{
    return a+b;
}
```

Przy wywołaniu procedur, w kodzie SPIM, funkcja *addthem* wymaga ramki w stosie (4 bajty) do zapisania wartości rejestru *t*0, którą należy przywrócić po zakończeniu obliczeń funkcji:

```
.text
main:  #założenia: a jest  w $t0, b jest  w $t1
    add $a0,$0, $t0   # Przekazanie argumentów
    add $a1,$0,$t1    # do rejestrów a0, a1
    jal addthem       # wywołanie funkcji addthem
    #Miejsce powrotu
    add $t3,$0,$v0    # gdy funkcja wywoływana zwroci
                      # wartość do $v0, jest ona przesłana do $t3
    syscall
addthem:
    addi $sp,$sp,-4   # zarezerwowanie ramki stosu
    sw $t0, 0($sp)    # zapis poprzedniej wartosci ($t0)
    add $t0,$a0,$a1   # instrukcja implementująca ciało funkcji
    add $v0,$0,$t0    # wynik
    lw $t0, 0($sp)    # ladowanie poprzedniej wartości
    addi $sp,$sp,4    # Zdejmij ramkę ze stosu
    jr $ra            # powrót
```

Przykład generowania kodu z użyciem AST (*Abstract Syntax Tree* – rysunek 15.6) i **CFG** (*Control Flow Graph* – rysunek 15.7).

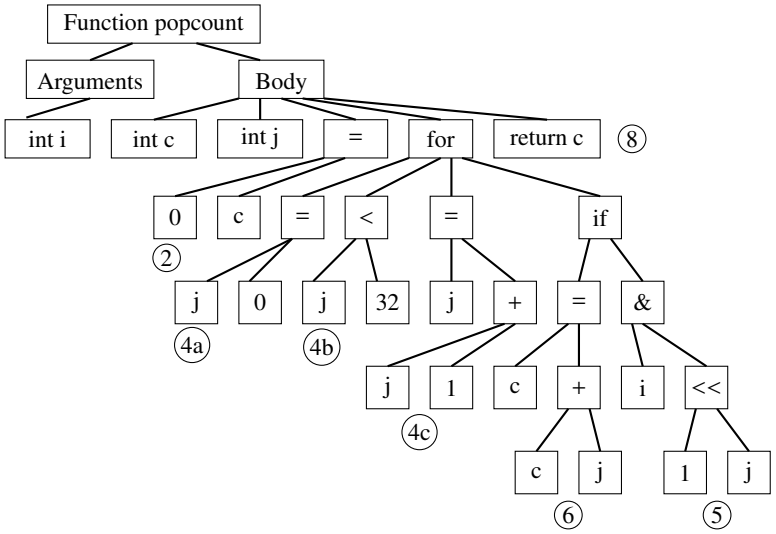
W kodzie MIPS w komentarzach umieszczone są numery linii ze źródłowego kodu w C, z których wyprowadzono daną linię kodu asemblera. Linia 4 ma 3 istotne elementy: a) inicjalizację licznika pętli, b) sprawdzenie warunku, c) inkrementację. Identyczne oznaczenia linii kodu znajdują sie w okręgach na rysunkach 15.6 i 15.7.

```
int popcount(int i){
    int c=0;
    int j;
    for (j=0 ; j<32 ;j++){
        if (i & (1 << j))
            c++;
    }
    return c
}
```

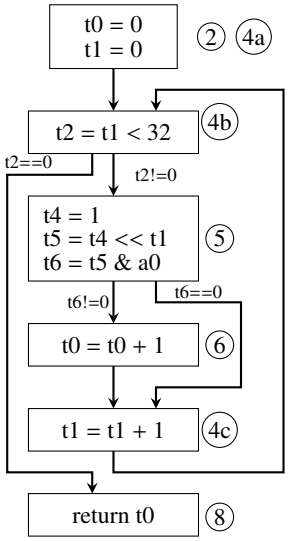
```
popcount:
        ori $v0, $zero, 0       #Linia 2
        ori $t1, $zero, 0       #4a
top:    slti $t2, $t1, 32       #4b
        beq $t2, $zero, end     #4b
        nop
```

```
7           addi $t3, $zero, 1      #5
8           sllv $t3, $t3, $t1      #5
9           and $t3, $a0, $t3       #5
10          beq $t3, $zero, notone  #5
11          nop
12          addi $v0, $v0, 1        #6
13  notone: beq $zero, $zero, top
14          addi $t1, $t1, 1        #4c
15  end:    jr $ra                  #8
16          nop
```

Nowe instrukcje w powyższym kodzie: *ori* – operator OR, *nop* – pusty operator, *sllv – shift left logical variable*.



Rys. 15.6. Drzewo składniowe (AST) dla kodu funkcji *popcount*



Rys. 15.7. CFG (*control flow graph*) dla kodu funkcji *popcount*

Kolejne dwa programy przedstawiają przykłady kodów w asemblerze. Dodawanie dwóch liczb:

```
# $t2 - used to hold the sum of the $t0 and $t1.
# $v0 - syscall number, and syscall return value.
# $a0 - syscall input parameter.
.text # Code area starts here
main:
    li $v0, 5       # read number into $v0
    syscall         # make the syscall read_int
    move $t0, $v0   # move the number read into $t0
    li $v0, 5       # read second number into $v0
    syscall         # make the syscall read_int
    move $t1, $v0   # move the number read into $t1
    add $t2, $t0, $t1
    move $a0, $t2 # move the number to print into $a0
    li $v0, 1   # load syscall print_int into $v0
    syscall     #
    li $v0, 10  # syscall code 10 is for exit
    syscall     #
# end of main
```

Dodawanie *N* liczb:

```
# Input: number of inputs, n, and n integers;   Output: Sum of integers
            .data                       # Data memory area.
prmpt1:     .asciiz "How many inputs? "
prmpt2:     .asciiz "Next input: "
sumtext:    .asciiz "The sum is "
            .text                   # Code area starts here
main:       li   $v0, 4         # Syscall to print prompt string
            la   $a0, prmpt1          # li and la are pseudo instr.
            syscall
            li   $v0, 5             # Syscall to read an integer
            syscall
            move $t0, $v0           # n stored in $t0

            li   $t1, 0                 # sum will be stored in $t1
while:      blez $t0, endwhile        # (pseudo instruction)
            li   $v0, 4             # syscal to print string
            la   $a0, prmpt2
            syscall
            li   $v0, 5
            syscall
            add  $t1, $t1, $v0      # Increase sum by new input
            sub  $t0, $t0, 1        # Decrement n
            j    while
endwhile:   li   $v0, 4              # syscall to print string
            la   $a0, sumtext
            syscall
            move $a0, $t1               # Syscall to print an integer
            li   $v0, 1
            syscall
            li   $v0, 10                # Syscall to exit
            syscall
```

Część II Zajęcia laboratoryjne

1. Wstęp. Analiza leksykalna

Zadania do samodzielnego wykonania:

1. Pobrać wszystkie pliki z przykładami na: `http://detox.wi.ps.pl/pb/tk/wszystkieZ.zip`.
2. Przetestować kompilację i uruchomienie kodu z katalogów *z1, z2, z3* (instrukcja na końcu rozdziału).
3. Uruchomić program poprzez podanie jako argumentu pliku inX.txt; pliki znajdują się w odpowiednich katalogach.
4. Dodatkowo przetestować pracę w trybie interaktywnym.
5. Zmodyfikować pliki wejściowe w *z2*, tak aby się „kompilowały" obydwa; ewentualnie można zmodyfikować plik z analizatorem (z2.l).
6. Zapoznać się z treścią *Makefile* w *z3*.
7. Pobrać emulator procesora MIPS – MARS na: `http://courses.missouristate.edu/KenVollmar/mars/download.htm` (alternatywnie QtSpim).
8. Uruchomić za pomocą emulatora MARS przykładowy program pobrany z adresu: `http://courses.missouristate.edu/KenVollmar/mars/CCSC-CP%20material/row-major.asm`

Zadanie domowe: Opracować projekt **własnego języka**. Łatwiej zrobić kompilator języka tradycyjnego; unikamy konstrukcji z języków ezoterycznych. Oprócz samych konstrukcji proszę przygotować kilka plików z programami w swoim języku (testujących poniższe konstrukcje). Poszczególne etapy rozbudowy prostego kompilatora:

- typy *int* i *double* – stałe (literały) i zmienne tych typów, wyrażenia arytmetyczne (=, +, -, *, /), prosty *if* (bez *else* i najwyżej jedno zagnieżdżenie), wypisywanie *int* i *double*, wpisywanie *int* i *double* z konsoli;
- pętla *for* lub *while*; typ *string* (tylko wypisywanie, brak operacji);
- tablice jednowymiarowe, złożony *if* (dużo zagnieżdżeń) z *else*;
- tablice wielowymiarowe;
- dynamiczna alokacja tablic jednowymiarowych lub proste funkcje (procedury bez parametrów) ewentualnie funkcje z parametrami i wartością zwracaną.

Wywoływanie kompilacji w pierwszych dwóch katalogach:

```
flex zX.l #powstaje plik lex.yy.c
gcc -c lex.yy.c #powstaje plik lex.yy.o
gcc lex.yy.o -o nazwa_mojego_kompilatora -ll # powstaje plik
    nazwa_mojego_kompilatora
```

Uruchamianie:

```
./nazwa_mojego_kompilatora #uruchomienie w trybie interaktywnym
./nazwa_mojego_kompilatora < in1.txt #uruchomienie z podaniem na stdin
    pliku in1.txt
./nazwa_mojego_kompilatora in1.txt #uruchomienie z podaniem w argv[1]
    in1.txt (trzeba obsluzyc w main)
```

Tryb interaktywny kończymy zawsze, naciskając:

```
Ctrl+D
```

Z pliku Makefile korzystamy, wpisując:

```
make
```

Wywołanie bisona (korzystamy z Makefile, ale trzeba wiedzieć, co się w nim dzieje):

```
bison -d def.y #powstaje plik def.tab.c i def.tab.h (w wersji C++ def.
    tab.cc i def.tab.hh)
gcc -c def.tab.c #powstaje plik def.tab.o
gcc lex.yy.o def.tab.o -o nazwa_mojego_kompilatora -ll # powstaje plik
    nazwa_mojego_kompilatora
```

2. Zwracanie leksemów

Zadania do samodzielnego wykonania:

1. Przygotować listę leksemów w analizatorze leksykalnym (plik zX.l).
2. Przygotować listę tokenów w części początkowej analizatora składniowego (plik def.y).
3. W analizatorze leksykalnym dopisać zwracanie tokenów – dla pojedynczych symboli jest to kod ASCII symbolu, dla słów kluczowych, liczb, identyfikatorów i symboli wieloznakowych jest kod tokenu.

Plik z opisem analizatora leksykalnego składa się z 3 części oddzielonych od siebie podwójnym znakiem %. Pierwsza część to część nagłówkowa, która pozwala na dołączenie plików nagłówkowych, przygotowanie prototypów funkcji wywoływanych w pozostałych częściach oraz zapisanie definicji do przetworzenia przez flexa. Druga część zawiera reguły przetwarzania tekstu składające się z wzorca i akcji, która jest wykonywana, jeżeli wzorzec zostanie dopasowany. W tej części (akcja) możemy również zwrócić kod tokenu do analizatora składniowego. Trzecia część to kod w języku C kopiowany bezpośrednio do pliku wynikowego w C.

Podstawe elementy reguł: Znaki [i] (nawiasy kwadratowe) pozwalają zapisać regułę dotyczącą jednego elementu wyrażenia regularnego. Jeżeli po takiej regule następuje znak: +, to reguła musi wystąpić w dopasowanym tekście w odpowiednim miejscu przynajmniej raz; może się powtarzać wielokrotnie (1–). Jeżeli następnym znakiem jest: *, to reguła może wystapić wielokrotnie, ale nie musi (0–). Na końcu listy reguł powinna się znaleźć kropka (.), która w tym kontekście oznacza brak wcześniejszego dopasowania, co powoduje błąd (nieznany leksem).

Plik z opisem analizatora składniowego jest zbudowany również z 3 sekcji: z sekcji nagłówkowej, z reguł i z akcji (tutaj nazywanych akcjami semantycznymi; na tym etapie nie będziemy jeszcze z nich korzystać); kod w C/C++. Na obecnym etapie musimy jedynie zdefiniować tokeny (przykłady poniżej) dla wszystkich słów kluczowych, liczb, identyfikatorów i symboli wieloznakowych (dla pojedynczych znaków kodem tokenu jest jego kod ASCII).

Po zdefiniowaniu tokenów i przetworzeniu pliku *def.y* za pomocą narzędzia bison możemy (bison wygeneruje nam m.in. plik nagłówkowy z liczbowymi kodami tokenów) zwracać kody tokenów z poziomu analizatora leksykalnego. Przykłady można znaleźć w katalogach *z4, z5 i z6*.

Definicja tokenu bez określania typu:

```
%token LEQ
```

Definicja kilku tokenów bez określania typu:

```
%token LEQ GEQ EQ
%token FOR INT DOUBLE
```

Definicja tokenu z określeniem typu:

```
%token <text> ID
```

Zwrócenie tokenu (całe reguły z akcjami w wersji podstawowej) z analizatora leksykalnego:

```
\=              {return '=';}
\<\=                {return LEQ;}
"int"           {return INT;}
[A-Za-z_][A-Za-z0-9_]*   {return ID;}
```

3. Gramatyka. Wartości semantyczne

Zadania do samodzielnego wykonania:

1. Przygotować gramatykę dla pojedynczego wyrażenia składającego się ze zmiennych, z liczb i operatorów (plik *def.y* – przykładowa gramatyka w katalogu *z5*).
2. Przekazać wartości semantyczne identyfikatorów i liczb (całkowitych i rzeczywistych) z analizatora leksykalnego do analizatora składniowego.
3. Zaimplementować funkcję *main* (tylko jedną wersję w *def.y*) z obsługą parametrów przekazywanych przy uruchamianiu (*argc, argv*).
4. Zapisać do pliku wartości semantyczne poszczególnych identyfikatorów i liczb oraz operatory w kolejności dopasowywania.
5. Przetestować działanie dla wyrażeń składających się z kilku (8–10) elementów.

Reguły gramatyki języka przetwarzane przez generator bison składają się:

– z nazwy symbolu nieterminalnego,
– ze znaku dwukropka,
– z definicji składającej się z symboli terminalnych i nieterminalnych.

Alternatywne definicje są od siebie oddzielone pionową kreską (|), a po ostatniej alternatywnej definicji (dla porządku) powinno się postawić średnik (;). Jako symbol startowy gramatyki jest wybierany symbol nieterminalny znajdujący się na początku, chyba że wskażemy go wprost (dyrektywa *%start*). Kolejność pozostałych reguł nie jest istotna, jednak lepiej byłoby, żeby reguły były zapisane w uporządkowany sposób. Do definiowania reguł jako symboli teminalnych używać należy uprzednio zdefiniowanych tokenów zdefiniowanych w pierwszej sekcji oraz symboli jednoznakowych (w apostrofach). Po każdej definicji (również po każdym jej fragmencie) można zapisać akcję (semantyczną) w postaci kodu C/C++, która zostanie wywołana, jeżeli reguła zostanie dopasowana.

Przykładowa definicja symboli nieterminalnych (wyrażenie, czynnik i składnik):

```
1  wyr
2      :wyr '+' skladnik        {printf(" + \n");}
3      |wyr '-' skladnik        {printf(" - \n");}
4      |skladnik                {printf("wyr \n");}
5      ;
6  skladnik
7      :skladnik '*' czynnik    {printf(" * \n");}
8      |skladnik '/' czynnik    {printf(" / \n");}
9      |czynnik                 {printf("skladnik \n");}
10     ;
11 czynnik
12     :ID                      {printf("zmienna\n");}
13     |LC                      {printf("liczba\n");}
14     |'(' wyr ')'             {printf("nawiasy\n");}
15     ;
```

Do przekazania wartości semantycznych z analizatora leksykalnego do składniowego nalezy użyć unii *yylval* (jej domyślna nazwa podana jest w kodzie generowanym przez flex). Pola tej unii definiuje się w pliku dla narzędzia

bison (np. *def.y*). Wartości leksemów dla liczb i identyfikatorów (dla pozostałych leksemów również) są przechowywane w zmiennej *yytext*.

Przykładowa definicja unii do przechowywania wartości semantycznych tokenów, których wartość semantyczną będziemy przekazywać (tokeny muszą być określonego typu):

```
%union
{
    char *text;
    int ival;
};
%token <text> ID
%token <ival> LC
```

W analizatorze składniowym do wartości semantycznej odwołać się można przy uzyciu symbolu $ i liczby. Liczba oznacza miejsce leksemu, którego wartość semantyczną chcemy otrzymać, w regule (najczęściej jest to $1, ale np. w przypadku deklaracji tablic może być to $2 lub $3).
Przykładowe odwołanie do wartości semantycznej identyfikatora:

```
czynnik
    :ID     {printf("id: %s\n", $1);}
    ;
```

4. Trójki. Rozbudowa gramatyki

Zadania do samodzielnego wykonania:

1. Zmodyfikować pliki źródłowe i plik *Makefile*, tak aby analizator składniowy i cały kompilator był kompilowany za pomocą kompilatora C++.
2. Zmodyfikować gramatykę tak, by można było kompilować program składający się z wielu wyrażeń (wielu linii).
3. Dopisać kod zapisujący trójki do pliku.

Modyfikacja w celu umożliwienia kompilacji za pomocą C++:

1. Zmieniamy nazwę pliku def.y na def.yy (dzięki temu bison wygeneruje nam pliki def.tab.cc i def.tab.hh).
2. W pliku Makefile:
   - zmieniamy odwołania do def.y na def.yy,
   - zmieniamy odwołania do def.tab.c na def.tab.cc,
   - dodajemy na początku nową zmienną: CPP=g++
   - zamieniamy w dwóch miejscach odwołanie do CC na CPP: kompilacja def.tab.cc, kompilacja całego programu (analizator leksykalny nadal kompilujemy za pomocą kompilatora C).
3. Do pliku def.yy dopisujemy w sekcji nagłówkowej:
   - *extern "C" int yylex();*,
   - *extern "C" int yyerror(const char *msg, ...);*,
   - *using namespace std;*,
   - Jeżeli mamy deklaracje *yyin* i *yyout*, to dopisujemy do nich również *extern* (bez C).
4. w pliku zx.l:
   - na początku zmieniamy dołączany plik nagłówkowy (def.tab.h na def.tab.hh),
   - na początku dodajemy *int yyerror(const char *msg, ...);* ; tak naprawdę zmieniamy tylko nagłówek funkcji *yyerror* – dodajemy *const*;
   - na końcu dodajemy *int yyerror(const char *msg, ...){*; tak naprawdę modyfikujemy tylko nagłówek funkcji *yyerror* – dodajemy const

Chcąc zdefiniować gramatykę do obsługi wielu wyrażeń, można się posłużyć analogią do wyrażeń arytmetycznych, które również mogą być dowolnie długie.

Jako trójki będziemy na tym etapie traktować wyrażenia składające się ze zmiennej *result* (w przyszłości numerowanej) oraz z dwóch argumentów i operatora. Na przykład dla wyrażenia $a = b + c * 9$; uzyskamy trzy trójki:

- $result = c\ 9\ *$,
- $result = b\ result\ +$,
- $result = a\ result\ =$.

To tylko przykładowa forma zapisu; różne kompilatory robią to swoimi sposobami. Aby uzyskać taką formę zapisu, należy liczby i identyfikatory odkładać na stosie (ten stos powinien przechowywać obiekty jakiejś struktury / klasy; oprócz wartości należy przechowywać również typ). Natomiast w akcjach semantycznych:

- dla operatorów arytmetycznych należy ściągać ze stosu odpowiednią liczbę argumentów (najczęściej 2);
- zapisać na stos zmienną przechowującą wynik;
- wszystkie te elementy (argumenty, operator i zmienną wyniku) zapisać do pliku.

Przypomnienie: klasa *std* :: *stack* ma metody:
- *push* – wstawienie wartości na szczyt stosu,
- *top* – pobranie wartości ze szczytu stosu,
- *pop* – usunięcie wartości ze szczytu stosu.

5. Generowanie kodu

Zadania do samodzielnego wykonania:

1. Dodać opcję kompilacji -std=c++11 do Makefile (w miejscach gdzie używamy g++).
2. Przygotować regułę dla symbolu nieterminalnego przedstawiającego przypisanie (jeżeli wcześniej tego nie zrobiliśmy).
3. Utworzyć tablicę symboli (zapisywać w niej wszystkie identyfikatory).
4. Generować zmienne tymczasowe do przechowywania wyników trójek.
5. Dla każdej trójki generować 4 linie kodu asemblera i przechowywać je w wektorze.
6. Dopisać kod zapisujący linie z wektora do pliku *yyout*, wywołać ten kod po *yyparse*.
7. Po *yyparse* zapisać tablicę symboli do pliku *symbols.txt*.
8. Zapisać symbole z tablicy symboli przed kodem w bloku danych.

Tablica symboli zawiera wszystkie identyfikatory, również identyfikatory zmiennych tymczasowych generowanych przez kompilator. Powinna to być tablica haszująca (w C++ *std* :: *map*). Tablica symboli pozwala przechować i wyszukiwać na podstawie identyfikatora informacje o zmiennych: typ zmiennej, miejsce przechowywania ewentualnie rozmiary tablicy. Należy ją pod koniec działania kompilatora zapisać do pliku *symbols.txt*.

Zmienne tymczasowe będą służyć do przechowywania w pamięci wyników pośrednich obliczeń. Kolejne zmienne powinny być numerowane. Nie powinno być możliwości użycia zmiennej o takiej samej nazwie jak zmienna tymczasowa w normalnym kodzie.

Przykładowy kod generowany dla trójki ma postać:

```
li $t0, 27
lw $t1, x
sub $t0, $t0, $t1
sw $t0, result15
```

Rejestry $\$t0$ – $\$t7$ to tak zwane rejestry tymczasowe. Wartość do takiego rejestru można wstawić za pomocą instrukcji (na razie) *li* lub *lw*. Instrukcja *li* wstawia wartość liczbową (bezpośrednią). Instrukcja *lw* wstawia wartość spod adresu wskazanego nazwą zmiennej. Instrukcja *sub* wykonuje odejmowanie, instrukcja *add* – dodawanie, *mul* – mnożenie, *div* – dzielenie, a *sw* wstawia wartość z rejestru do komórki pamięci. W przypadku operacji arytmetycznych operacja jest wykonywana na dwóch ostatnich rejestrach, a wynik jest przechowywany w pierwszym rejestrze.

Uogólniony kod generowany dla trójki ma postać (podkreślenia wskazują miejsca do wypełnienia):

```
l_ $t0, __
l_ $t1, __
___ $t0, $t0, $t1
sw $t0, ____
```

Kod dla przypisania można skrócić (formę skróconą trzeba wymyślić samemu).

Część nagłówkowa (blok danych) służyć będzie do zarezerwowania miejsca na zmienne (w przyszłości również stałe). Blok danych zaczyna się od dyrektywy *.data*.

Format bloku danych:

```
nazwa: typ wartosc
```

Przykłady:

```
        x:      .word   0
        arr:        .space  40
        napis:      .asciiz  "Napis na ekran"
        f:      .float 3.14
```

Kod zaczyna się od dyrektywy *.text*. Komentarz w kodzie asemblera jest oznaczany znakiem: #.

Przykładowy kod dla wyrażenia ($x = 3; z = 5 + x * 2;$):

```
.data
    x:  .word   0
    z:  .word   0
  result1:  .word   0
  result2:  .word   0
.text
    li $t0, 3
    sw $t0, x
    lw $t0, x
    li $t1, 2
    mul $t0, $t0, $t1
    sw $t0, result1

    li $t0, 5
    lw $t1, result1
    add $t0, $t0, $t1
    sw $t0, result2

    lw $t0, result2
    sw $t0, z
```

6. Wejście / wyjście. Instrukcje warunkowe

Zadania do samodzielnego wykonania:

1. Dodać kompilację instrukcji do wprowadzania i wypisywania wartości:
   - reguły gramatyki,
   - generowanie kodu wynikowego.
2. Dodać kompilację instrukcji warunkowych:
   - reguły gramatyki,
   - generowanie kodu wynikowego.

Wypisywanie na ekran i pobieranie danych z konsoli w MIPS realizuje się za pomocą wywołań systemowych o określonych identyfikatorach.

Przykładowy kod wyświetlający liczbę całkowitą na ekran:

```
li   $v0, 1
li   $a0, 42
syscall
```

Przykładowy kod pobierający liczbę całkowitą do zmiennej:

```
.data
x: .word 0
.text
    li $v0, 5
    syscall
    sw $v0, x
```

Przykładowy kod wypisujący łańcuch znaków:

```
.data
str: .asciiz "Tekst do wypisania."
.text
    li $v0, 4
    la $a0, str
    syscall
```

W rejestrze $\$v0$ wpisujemy identyfikator wywołania systemowego. Poniżej przedstawiono listę przydatnych wywołań wraz z rejestrami, w których jest wynik lub z rejestrami, do których należy przekazać wartość (identyfikator wywołania, nazwę, parametr lub wartość zwracaną):

id	Nazwa	Parametr lub wartość zwracana
1	*print integer*	$\$a0$ – liczba całkowita do wypisania
2	*print float*	$\$f12$ – liczba zmiennoprzecinkowa do wypisania
4	*print string*	$\$a0$ – adres łańcucha znaków zakończonego zerem (asciiz) do wypisania
5	*read integer*	$\$v0$ – odczytana z klawiatury liczba całkowita
6	*read float*	$\$f0$ – odczytana z klawiatury liczba zmiennoprzecinkowa

Instrukcja warunkowa (w prostszej formie) składa się z dwóch elementów – z części z wyrażeniem warunkowym i z bloku kodu wykonywanego przy spełnionym warunku. Reguły dla wersji z *else* należy opracować samodzielnie.

Wyjściowe reguły gramatyki dla instrukcji warunkowej:

```
if_expr
    :if_begin code_block  {
                              gen_etykiety_koncowej();
                          }
if_begin
    :IF '(' cond_expr ')' {
                              gen_warunku_i_skoku();
                          }
```

Etykiety w kodzie MIPS oznacza się za pomocą nazwy i symbolu dwukropka, skok do etykiety odbywa się przez użycie instrukcji skoku i nazwy etykiety:

```
    b  LBL42
    li $v0, 6
LBL42:
    li $v0, 5
LBL43:    syscall
    sw $v0, x
```

Każda nowa instrukcja warunkowa (również *else*; później również instrukcje pętli) powoduje wygenerowanie nowej etykiety (z kolejnym numerem). Etykiety należy wstawić na stos (oddzielny stos etykiet). W momencie dopasowania (akcja semantyczna dla $f_e xpr$) całej konstrukcji warunkowej należy zdjąć etykietę ze stosu i wstawić ją do kodu (wstawiać z dwukropkiem, przechowywać bez niego).

Zestaw przykładowych instrukcji skoków; można użyć np. instrukcji obliczających wyrażenie (*seq* itd.):

```
beq $t0,$t1,label #skok gdy rowne
bne $t0,$t1,label #skok gdy nie rowne
bge $t0,$t1,label #skok gdy $t0 wiekszy, rowny
bgt $t0,$t1,label #skok gdy $t0 wiekszy
ble $t0,$t1,label #skok gdy $t0 mniejszy, rowny
blt $t0,$t1,label #skok gdy $t0 mniejszy
```

Dla kodu:

```
z=3;
if (x < 5+4 )
{
    y = 2+4;
    z = 3*3;
}
z = z*3;
```

Należy wygenerować następujący kod (symbolicznie):

```
z=3;
if (!(x < 5+4) )
goto LBL5:
{
    y = 2+4;
    z = 3*3;
}
LBL5:z = z*3;
```

W kolejnym kroku należy wygenerować następujący kod (mnemoniki) zapisany symbolicznie:

```
li $t0,3
sw $t0,z
obliczenie 5+4 (4 linie)
lw $t2, x
lw $t3, result23 #wynik(5+4)
bge $t2,$t3,LBL5:
#6 linii dla y=2+4
#6 linii dla z = 3*3;
LBL5: lw $t0,z
#5 pozostalych linii dla z = z*3;
```

Uwaga: Jest to jedna z możliwych metoda generowania kodu dla instrukcji warunkowych; można np. dodać dodatkową etykietę, dzięki której będzie można skorzystać z naturalnych (odwrotnych do wyżej użytych) instrukcji skoku.

7. Pętle. Tablice jednowymiarowe

Zadania do samodzielnego wykonania:

1. Dodać kompilację instrukcji pętli,
   - reguły gramatyki,
   - generowanie kodu wynikowego.
2. Dodać kompilację tworzenia i używania statycznych tablic jednowymiarowych,
   - reguły gramatyki,
   - generowanie kodu wynikowego.

Pętle. Można je realizować za pomocą skoku warunkowego i licznika. Licznik jest zmienną. Wartość tej zmiennej należy modyfikować w odpowiednim momencie (koniec pętli, początek pętli), z wyjątkiem pierwszej iteracji.

Wyjściowe reguły gramatyki dla instrukcji pętli podobnej do pętli w C (wszystkie elementy wymagane):

```
for_expr
    :for_begin code_block
    {gen_etykiety_koncowej_i_skoku();}
for_begin
    :FOR '(' init_expr ';' cond_expr ';' inc_expr ')' {
       gen_warunku_i_skoku();}
```

Dla następującego kodu pętli:

```
for (i = 0 ; i < 10 ; ++i )
{
    z = z + i ;
}
z = z*3;
```

należy wygenerowac następujący kod (symbolicznie):

```
 i=0;
 goto LBL5
LBL6:
 ++i;
LBL5:
 if(i>=10)
   goto LBL7:
 {
     z = z + i ;
 }
 goto LBL6:
LBL7:
 z = z*3;
```

Dzięki powyższej konstrukcji nie ma potrzeby zapamiętywania kodu wyrażenia warunkowego i wyrażenia inkrementującego (ceną za takie uproszczenie jest dołożenie dodatkowej etykiety). Etykiety są ponumerowane w kolejności ich pojawiania się (pierwszy pojawia się skok do etykiety LBL5). Etykiety (w przykładzie LBL6 i LBL7) należy zapamiętać na stosie etykiet. W akcji semantycznej, wywoływanej po dopasowaniu całej konstrukcji (wraz z blokiem kodu) pętli, należy zdjąć etykiety i wygenerować instrukcję skoku do drugiej etykiety oraz umieścić pierwszą etykietę w kodzie (z dwukropkiem).

Wygenerowany kod assemblera (mnemoniki):

```
 li $t0,0
 sw $t0,x
 b LBL5
LBL6:
 #te 4 linie mozna prosciej zapisac (ale nie trzeba)
 lw $t0, i
 li $t1, 1
 add $t0, $t0, $t1
 sw $t0, i
LBL5:
 lw $t2, i
 li $t3, 10
 bge $t2, $t3, LBL7
 lw $t0, z
 lw $t1, i
 add $t0, $t0, $t1
 sw $t0, result1
 lw $t0, result1
 sw $t0, z
 b LBL6
LBL7:
 lw $t0, z
 li $t1, 3
 mul $t0, $t0, $t1
 sw $t0, result2
 lw $t0, result2
 sw $t0, z
```

Uwaga: To jest jedna z możliwych metod generowania kodu dla instrukcji pętli. Możliwych modyfikacji jest więcej niż przy instrukcjach warunkowych. Można np. zmienić położenie skoków, zapamiętać inne wartości.

Obsługa **tablic jednowymiarowych** obejmuje zapamiętanie rozmiaru tablicy i odwołanie do indeksowanego elementu tablicy.

Uwaga: Odwołania mogą występować po lewej i prawej stronie wyrażeń.

Wyjściowe reguły gramatyki dla deklaracji (uproszczone w stosunku do C – w C rozmiar można określić wyrażeniem stałym i deklarować wiele zmiennych):

```
arr_decl
 :INT ID '[' LC ']' ';'
       {tablica_symboli[$2], typ=ARRI, rozmiar=$4; }
 :FLOAT ID '[' LC ']' ';'
       {tablica_symboli[$2], typ=ARRF, rozmiar=$4; }
```

Wyjściowe reguły gramatyki dla indeksowania:

```
arr_expr
    :ID '[' wyr ']'
```

Rozmiar tablicy w MIPS podajemy po dwukropku (lepiej użyć *.word*). Przy odwoływaniu się do elementu tablicy należy pomnożyć przez 4 wartość przesunięcia względem adresu początkowego; adres początku tablicy ładujemy za pomocą *la*. Wartość pod adres wpisujemy lub pobieramy za pomocą nawiasów okrągłych (np. *sw* $*t*0, ($*t*4)).

Dla kodu:

```
int a[10];
a[3] = 2;
x = a[2+1];
```

należy wygenerować następujący kod (mnemoniki):

```
.data
    a: .word 0:10
    x: .word 0
    result1: .word 0
.text
li $t0, 2
la $t4, a
li $t5, 3
mul $t5, $t5, 4
add $t4, $t4, $t5
sw $t0, ($t4)

li $t0, 2
li $t1, 1
add $t0, $t0, $t1
sw $t0, result1
la $t4, a
lw $t5, result1
mul $t5, $t5, 4
add $t4, $t4, $t5
lw $t0, ($t4)
sw $t0, x
```

8. Liczby zmiennoprzecinkowe. Tablice wielowymiarowe

Zadania do samodzielnego wykonania:

1. Dodać obsługę liczb zmiennoprzecinkowych:
   - generowanie kodu obliczeń,
   - konwersja lub zgłoszenie błędu.
2. Dodać obsługę statycznych tablic wielowymiarowych:
   - reguły gramatyki,
   - tworzenie wpisu w tablicy symboli,
   - obliczanie indeksu.

Obsługę liczb zmiennoprzecinkowych w MIPS realizuje się za pomocą innych rejestrów oraz innego zestawu instrukcji niż w przypadku ich całkowitych odpowiedników. Ponadto należy uprościć implementację odwołań do stałych zmiennoprzecinkowych poprzez traktowanie ich jak zmiennych. Rejestry zmiennoprzecinkowe, które są odpowiednikami rejestrów $\$t0$ – $\$t7$ to $\$f0$ – $\$f31$. Do przechowywania liczb typu *float* używa się pojedynczego rejestru, do przechowywania liczb typu *double* używa się pary rejestrów. Wymaganie w kompilatorze dotyczy tylko liczb typu *float*, a więc dalej opisano tylko instrukcje ich dotyczące. Do załadowania wartości zmiennej do rejestru służy instrukcja: *l.s rejestr*, *zmienna*. Do skopiowania wartości z rejestru do zmiennej służy instrukcja: *s.s rejestr*, *zmienna*. Do wykonywania operacji arytmetycznych służą operacje:

- *add.s rejestr_wynik*, *rejestr_arg*1, *rejestr_arg*2 – dodawanie,
- *sub.s rejestr_wynik*, *rejestr_arg*1, *rejestr_arg*2 – odejmowanie,
- *mul.s rejestr_wynik*, *rejestr_arg*1, *rejestr_arg*2 – mnożenie,
- *div.s rejestr_wynik*, *rejestr_arg*1, *rejestr_arg*2 – dzielenie,

Dla przykładowego kodu dodającego dwie liczby zmiennoprzecinkowe:

```
z=3.14+6.28;
y=3.14+5.12;
```

Należy wygenerować (stała może wystąpić w sekcji danych raz lub wielokrotnie; jeżeli występuje tylko raz, to jej wszystkie wystąpienia zastępujemy tym samym identyfikatorem):

```
.data
  z: .float 0
  float_var1: .float 3.14
  float_var2: .float 6.28
  y: .float 0
  float_var3: .float 5.12
  tmp1: .float 0
  tmp2: .float 0
.text
  l.s  $f0, float_var1
  l.s  $f1, float_var2
  add.s $f0, $f0, $f1
  s.s $f0, tmp1
```

```
14
15    l.s $f0, tmp1
16    s.s $f0, z
17
18    l.s  $f0, float_var1
19    l.s  $f1, float_var3
20    add.s $f0, $f0, $f1
21    s.s $f0, tmp2
22
23    l.s $f0, tmp2
24    s.s $f0, y
```

Jeżeli język zakłada możliwość konwersji pomiędzy zmiennymi typów zmiennoprzecinkowych i całkowitych, należy załadować wartości do rejestru, a następnie wywołać konwersję. Konwersję w obie strony wykonuje się w rejestrze zmiennoprzecinkowym.

Konwersja z wartości całkowitej na wartość zmiennoprzecinkową:

```
1  .text
2    li $t0, 10
3    mtc1 $t0, $f0
4    cvt.s.w $f1, $f0
```

Konwersja z wartości zmiennoprzecinkowej na wartość całowitą:

```
1  .data
2   float_var1: .float 3.14
3  .text
4    li $t0, 10
5    mtc1 $t0, $f0
6    cvt.s.w $f1, $f0
7    l.s $f2, float_var1
8    add.s $f1, $f1, $f2
9    cvt.w.s $f0, $f1
10   mfc1 $t0, $f0
```

Obsługa tablic wielowymiarowych obejmuje odczyt rozmiarów tablicy (np. z deklaracji) i odwołanie do indeksowanych elementów tablicy.

Wyjściowe reguły gramatyki dla tablic wielowymiarowych (propozycja):

```
1  arr_decl
2      :arr_start dim_decl {;}
3      ;
4  arr_start
5      : arr_type ID
6      ;
7  dim_decl
8      : '[' size_const ']' {;}
9      ;
10 size_const
11     : size_const ',' size_value {;}
12     | size_value    {;}
13     ;
14 size_value
15     : LC {;}
16     ;
```

W przypadku tablic wielowymiarowych należy zapamiętać w tablicy symboli do późniejszego przetwarzania długości poszczególnych wymiarów tablicy. Można (co jest bardzo przydatne) zapamiętać rozmiary pojedynczego elementu w przypadku poszczególnych wymiarów.

Dla deklaracji:

```
int a[4,3,5];
```

Informacja o tablicy powinna zawierać takie informacje:

```
dims: [4,3,5]
sizes: [15,5,1]
```

Przy generowaniu odwołań do tablic postępowanie wygląda podobnie jak przy tablicach jednowymiarowych, z tą różnicą, że należy odpowiednie odwołania mnożyć przez wartość z informacji o rozmiarach (sizes). Należy również pamiętać o mnożeniu wartości przez 4 (można te mnożenia połączyć w jedno mnożenie).

9. Funkcje. Dynamiczne tablice

Zadania do samodzielnego wykonania:

1. Dodać kompilację deklaracji i wywołania prostych funkcji:
   - reguły gramatyki,
   - generowanie kodu wynikowego.
2. Dodać kompilację instrukcji tworzenia dynamicznych tablic jednowymiarowych:
   - reguły gramatyki,
   - generowanie kodu wynikowego.

Funkcje w MIPS realizuje się za pomocą instrukcji *jal*, *jr* i rejestru $*ra*.

Przykładowy kod wywołujący prostą funkcję (ostatnia instrukcja *syscall* kończy działanie programu, żeby nie trzeba było wykonywać jeszcze raz kodu zawartego w funkcji):

```
.text
    li $t0, 42
    jal myfoo
    li $v0, 10
    syscall

myfoo:
    li $t0, 88
    jr $ra
```

Alternatywna wersja z funkcją *main* (trzeba do niej samodzielnie „skoczyć”):

```
.text
    b main
myfoo:
    li $t0, 88
    jr $ra

main:
    li $t0, 42
    jal myfoo
    li $t1, 88
```

Reguły gramatyki dla funkcji należy opracować samodzielnie (zgodnie z projektem własnego języka). Możliwe jest wykonywanie na końcu działania kompilatora dodatkowej pętli iterującej po wygenerowanym kodzie i liście funkcji, a także zamiana tych nazw na właściwe etykiety. Można też etykiety wstawiać od razu w trakcie kompilacji; na końcu należy tylko zweryfikować, czy wszystkie wywoływane funkcje istnieją. Ewentualne parametry do funkcji można przekazywać poprzez rejestry $*a*0 i $*a*1; wartość można zwracać poprzez rejestr $*v*0. W przypadku

kompilatora, budowanego zgodnie z wcześniej uproszczonymi założeniami, wartości nie są przechowywane w rejestrach, a tylko do nich wstawiane na czas wykonywania obliczeń. Nie jest również konieczne przechowywanie wartości rejestrów na stosie. Jeżeli jednak zaszłaby taka konieczność, funkcja na starcie przesuwa wartość w rejestrze $\$sp$ o -4 (rozmiar przechowywanego rejestru) i ładuje pod adres wskazywany przez ten rejestr wartość z rejestru do zapamiętania. Na końcu funkcji znajduje się „lustrzany" kod, który pobiera wartości rejestrów ze stosu i zwiększa wartość w rejestrze $\$sp$.

Dynamiczne tablice w MIPS tworzy się za pomocą wywołania systemowego sbrk. Reguły gramatyki należy opracować samodzielnie.

Dla przykładowego kodu w C++:

```
int n=20;
int *x = new int[10];
int *y = new int[n];
```

Należy wygenerować kod:

```
.data
    n:  .word 0
    x:  .word 0
    y:  .word 0
.text
    li $t0, 20
    sw $t0, n

    li $v0, 9
    li $a0, 10
    syscall
    sw $v0, x

    li $v0, 9
    lw $a0, n
    syscall
    sw $v0, y
```

Odwoływanie się do komórek tak utworzonych tablic jest identyczne jak odwoływanie się do tablic o statycznym rozmiarze.

10. Struktury

Zadanie do samodzielnego wykonania:
Dodać kompilację instrukcji do obsługi struktur (ew. klas):
– reguły gramatyki,
– generowanie kodu wynikowego.

Struktury w MIPS realizuje się poprzez odpowiednie umieszczenie zmiennej będącej strukturą w pamięci. Dla każdego elementu należącego do struktury, w tym także dla struktur wchodzących w skład struktury, należy określić liczbę zajmowanych komórek. W przypadku obsługi dynamicznie alokowanych strutur konieczne jest skorzystanie z funkcji przerwania *sbrk*, jak przy dynamicznie alokowanych tablicach. W przypadku statycznej zmiennej będącej strukturą należy w bloku danych zarezerwować blok pamięci o odpowiednim rozmiarze (dla każdej zmiennej będącej strukturą). Ponadto przydatne jest zachowanie przesunięcia poszczególnych pól względem początku struktury (przykładowa realizacja: mapa odwzorowująca nazwy elementów struktury na przesunięcie). Przy odwołaniu do elementu struktury należy postępować podobnie jak przy tablicach – do adresu bazowego dodajemy przesunięcie i odwołujemy się do komórki pod adresem będącym wynikiem tej operacji.

Dla przykładowego kodu deklarującego strukturę:

```
struct example
{
    int x;
    float z;
    int y;
};
struct example e1;
struct example e2;
struct example e3;
```

zostanie wygenerowany następujący fragment kodu odpowiedzialnego za rezerwację pamięci:

```
.data
    e1: .space 12
    e2: .space 12
    e3: .space 12
```

Dla kilku przypadków użycia tej struktury:

```
e1.y = 42;
e1.x = 10;

e2.y = 242;
e2.x = 210;

e3.y = 135;
e3.x = 145;
```

należy wygenerować kod:

```
.text
    li $t0, 42
    la $t4, e1
    add $t4, $t4, 8
    sw $t0, ($t4)
    li $t0, 10
    la $t4, e1
    add $t4, $t4, 0
    sw $t0, ($t4)

    li $t0, 242
    la $t4, e2
    add $t4, $t4, 8
    sw $t0, 8($t4)
    li $t0, 210
    la $t4, e2
    add $t4, $t4, 0
    sw $t0, ($t4)

    li $t0, 135
    la $t4, e3
    add $t4, $t4, 8
    sw $t0, 8($t4)
    li $t0, 145
    la $t4, e3
    add $t4, $t4, 0
    sw $t0, ($t4)
```

W przypadku struktur można odwołanie do elementu uprościć (ponieważ nie ma potrzeby indeksowania wyrażeniem):

```
.text
    li $t0, 42
    la $t4, e1
    sw $t0, 8($t4)
    li $t0, 10
    la $t4, e1
    sw $t0, 0($t4)

    li $t0, 242
    la $t4, e2
    sw $t0, 8($t4)
    li $t0, 210
    la $t4, e2
    sw $t0, 0($t4)

    li $t0, 135
    la $t4, e3
    sw $t0, 8($t4)
    li $t0, 145
    la $t4, e3
    sw $t0, 0($t4)
```

Bibliografia

1. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd edition, [b.m.], Addison Wesley, 2006.
2. D. Grune, C.J.H. Jacobs, *Parsing Techniques*, New York, Springer-Verlag, 2008.
3. S. Muchnik, *Advanced Compiler Design and Implementation*, San Francisco, Morgan Kaufmann Publishers, 1997.
4. J.R. Levine, T. Mason, D. Brown, *lex & yacc*, 2nd edition, Sebastopol, O'Reilly Media, 1992.
5. J.E.F. Friedl, *Mastering Regular Expressions*, 3rd edition, Sebastopol, O'Reilly Media, 2006.
6. J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd edition, New York, ACM, 2001.
7. D. Grune, K.van Reeuwijk, H.E. Bal, C.J.H. Jacobs, K.Langendoen, *Modern Compiler Design*, 2nd edition, New York, Springer, 2012.
8. T. Niemann, *A Compact Guide to Lex & Yacc*, http://epaperpress.com/lexandyacc/, dostęp: 22.10.2018.
9. *The Lex & Yacc Page*, http://dinosaur.compilertools.net/, dostęp: 22.10.2018.