PARADYGMATY I JĘZYKI PROGRAMOWANIA

Analiza leksykalna i syntaktyczna.

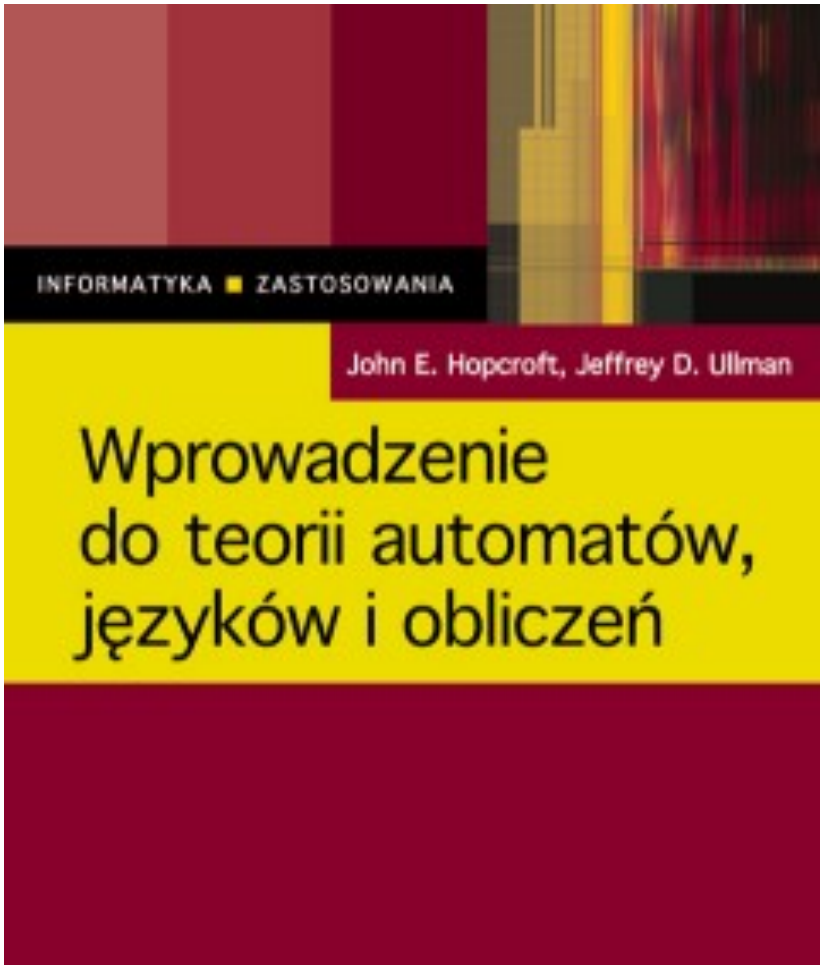
w-5

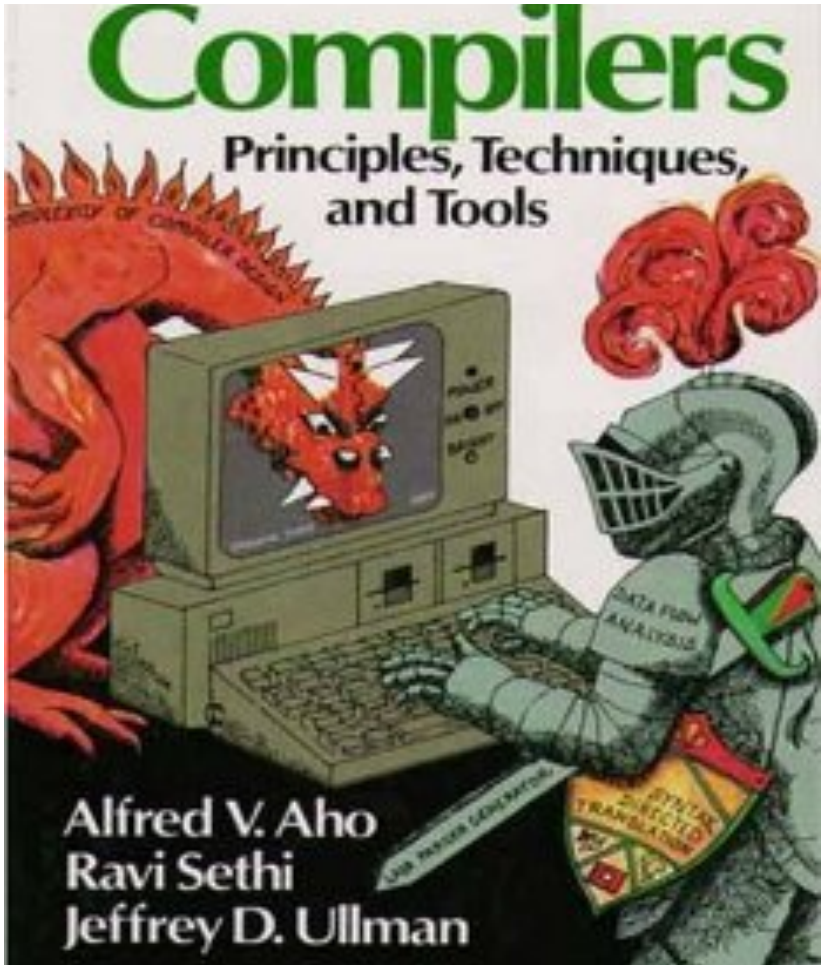
Treść

- Wstęp
- Analiza leksykalna
- Analiza syntaktyczna
- Rekurencja zstępująca
- Parsery LL
- Parsery LL sterowane tablicą

Wstęp

dobra książka





Wstęp

- Analiza programów źródłowych jest podstawą każdego systemu translacji kodu źródłowego
- Prawie cała analiza syntaktyczna programu opiera się na formalnym opisie składni w metajęzyku
- BNF/EBNF

Analiza syntaktyczna

- Część syntaktyczna każdego procesora języka prawie zawsze składa się z dwóch kawałków:
  - niskopoziomowego, zwanego *analizatorem leksykalnym* (skaner) (matematycznie jest to automat skończony oparty na gramatyce regularnej)
  - wyższego poziomu, zwanego *analizatorem syntaktycznym* (parser) (matematycznie jest to PDA czyli *push-down automaton, oparty na gramatyce bezkontekstowej lub EBNF*)

Zalety stosowania BNF

- Prosty i jasny sposób opisu składni
- Parser może bezpośrednio korzystać z BNF
- Parsery oparte na BNF są proste w obsłudze

Powody podziału analizy na leksykalną syntaktyczną

- *Prostota*. Analizatory leksykalne są prostsze i wydzielenie skanera upraszcza analizator leksykalny (kod)
- *Wydajność*. Wydzielanie skanera pozwala zoptymalizować analizę leksykalną
- *Przenośność*. Skaner (jego część) nie są przenośne podczas gdy parsery są programami przenośnymi.

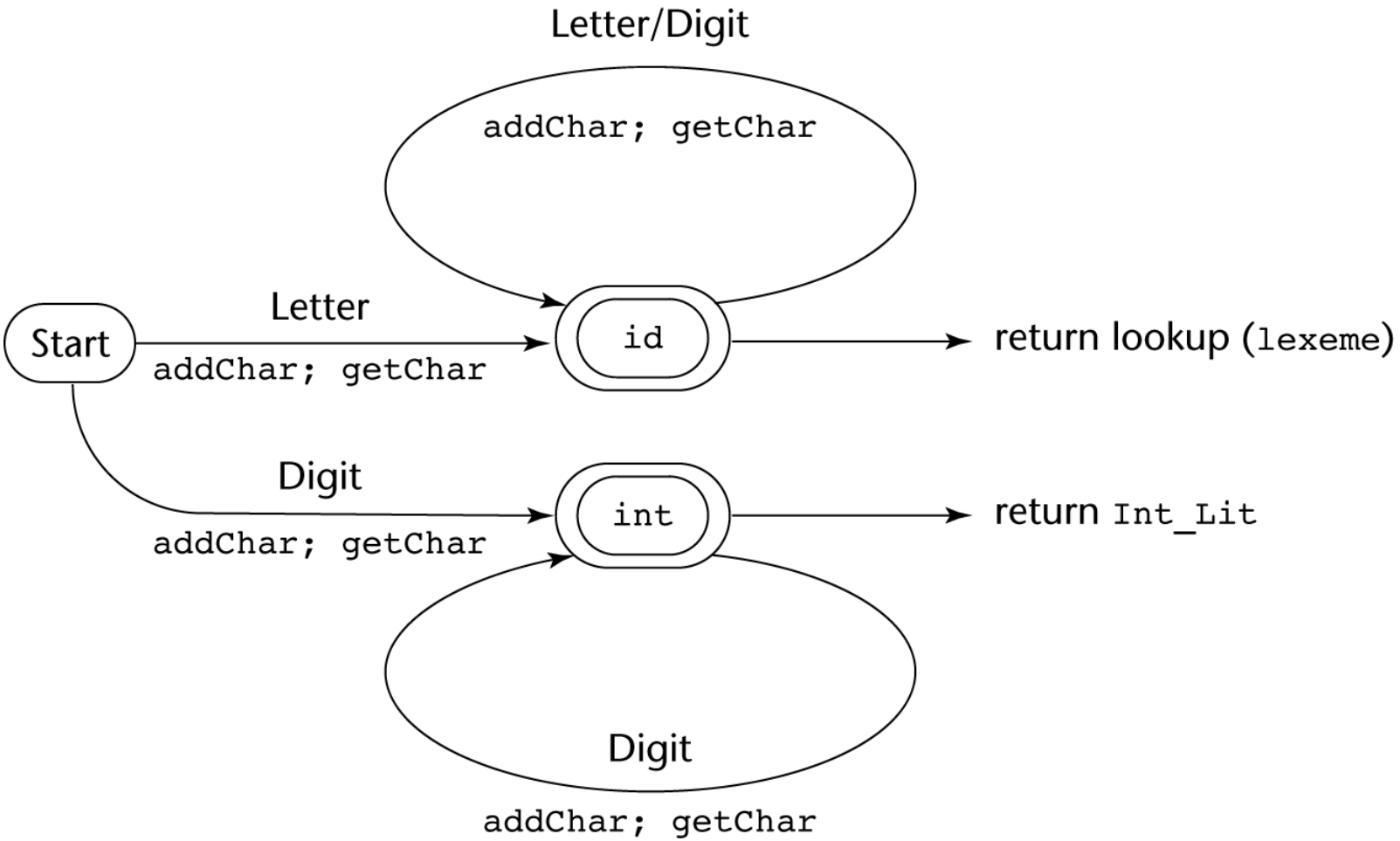
Analiza leksykalna

- Analizator leksykalny (skaner) dopasowuje wejściowe napisy do wzorców
- Jest pierwszym ogniwem w ciągu przetwarzania kodu, dostarczając danych dla analizatora składni (parsera)
- Identyfikuje napisy (leksemy) dzieląc je na kategorie słowne (tokeny)
  - np. **`suma`** jest leksemem, a jego kategorią może być **`IDENT;`** **`+`** jest operatorem - **`OP_PLUS;`** itd.

Analiza leksykalna

- Skaner jest najczęściej *funkcją*, którą wywołuje parser jeśli potrzebuje następnego tokenu
- Trzy różne sposoby budowy skanera
  - Podać formalny opis tokenów i wykorzystać program narzędziowy, który napisze analizator na podstawie tego opisu.
  - Zaprojektować diagram stanów opisujący tokeny i napisać program oparty na tym diagramie.
  - Zaprojektować diagram stanów opisujący tokeny i ręcznie zbudować program korzystający z tablicy opartej na diagramie stanów.

Przykład: Diagram stanów



Zadania analizy składniowej

Parsowanie

Zadania analizy składni (parser) 1

- Cele analizy składniowej
  - znalezienie błędów składni, ich sygnalizacja i powrót parsera do dalszej pracy
  - Budowa drzewa składni lub jego części dla analizowanego programu

Zadania analizy składni (parser) 2

- Dwie kategorie parserów
  - *Top-down* – zbudować drzewo składni, startując z głównego węzła (korzenia drzewa)
    - Kolejność odpowiada lewostronnym produkcjom
    - Drzewo budowane jest w zwykłej kolejności
  - *Bottom-up* – zbudować drzewo składni, startując od liści
    - Kolejność jest odwrotna i odpowiada produkcjom prawostronnym
- Najczęściej parsery czytają jeden token z wejścia “do przodu”

Zadania analizy składni (parser) 3

- OZNACZENIA:
  - T – symbole terminalne;
  - N – symbole nieterminalne
    - T – małe litery łacińskie z początku alfabetu (a, b, c)
    - N – duże litery łacińskie z początku alfabetu (A, B, C)
    - T|N – koniec alfabetu łacińskiego (W, X, Y, Z)
    - ciągi T – małe litery łacińskie z końca alfabetu (x, y, z)
    - napisy mieszane (T|N) – małe l. greckie (α, β, γ, δ)
  - =>* – ciąg produkcji; G – gramatyka;

Zadania analizy składni (parser) 4

- Parsery top-down
  - Jeśli mamy formę zdaniową xAα wówczas parser, który używa metody *lewostronnej*, musi wybrać właściwą regułę A by pobrać następną formę zdaniową, używając w tym celu tylko pierwszego tokenu z produkcji A. Np. reguły: A=>bB | cBb | a; parser musi wybrać jedną z trzech możliwości: xbBα, xcBbα lub xaα. Jest to decyzyjny problem parsera typu **top-down**.
- Parsery z rodziny LL (pierwsze L oznacza, że *wejście* jest przeglądane *od lewej strony (left to right)*, drugie L oznacza *lewostronny (leftmost first)*)

Zadania analizy składni (parser) 5

- Parsery typu **bottom-up**

  Parser rozpoczyna od ciągu terminali (liści drzewa parsowania). Jeśli znajdzie prawostronną formę zdaniową α zastępuje ją lewostronnym odpowiednikiem - następuje redukcja, tzn. wyodrębniony ciąg zostaje zastąpiony lewą stroną właściwej reguły gramatycznej. Cel polega na przeprowadzeniu wszystkich redukcji aż do symbolu startowego gramatyki.

  - Np. reguły G są: S=>aAc, A=>aA | b; produkcja S=>aAc=>aaAc=>aabc. Parser startuje z wejścia aabc i chce znaleźć odpowiednik w regułach. Tutaj łatwo znajduje b, które powstaje z reguły A=>b. Parser, zastępując b przez A, otrzymuje aaAc; tutaj znajdzie aA, które pochodzi z produkcji A=>aA i ma teraz aAc, co otrzymuje się z pierwszej produkcji S=>aAc.

- Rodzina parserów LR (R oznacza *pierwszy z prawej strony; rightmost first)*

Parsery z rekurencją zstępującą

LL

Przykład

Rekurencja zstępująca

- Wejście (ciąg napisów)

A , B , C ;

- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Przykład

- 1 `lista`

- Wejście (ciąg napisów)

A , B , C ;

- Gramatyka

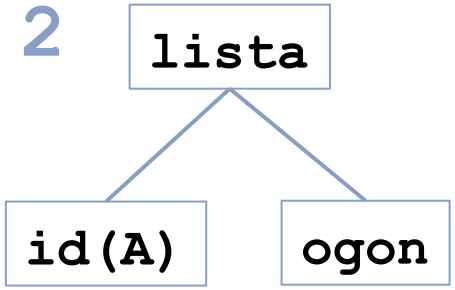
```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Przykład

- Wejście (ciąg napisów)

A , B , C ;

- 2



- Gramatyka

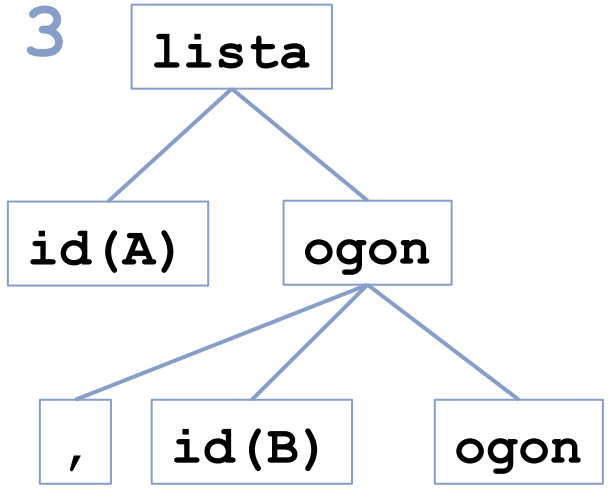
```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Przykład

- Wejście (ciąg napisów)

A , B , C ;

- 3



- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Przykład

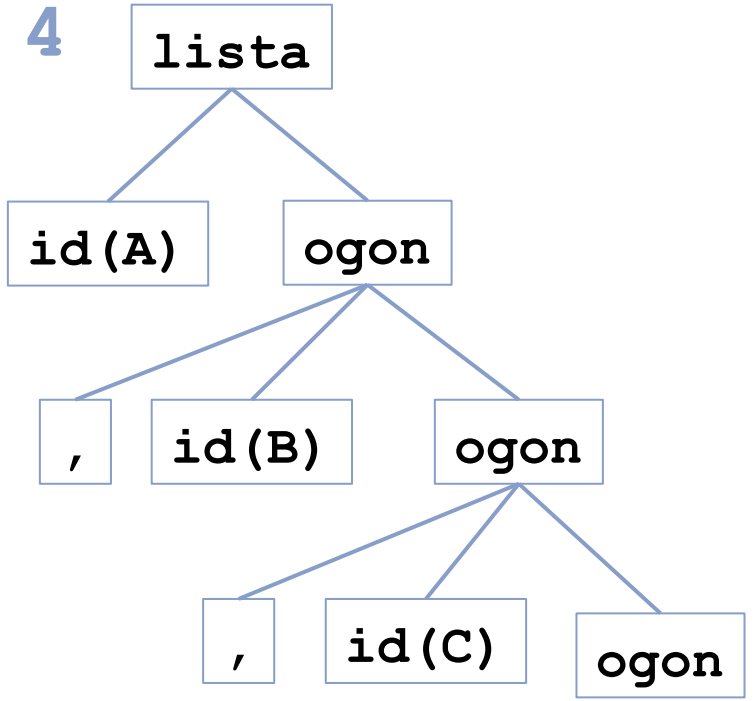
- Wejście (ciąg napisów)

A , B , C ;

- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

- 4



Przykład

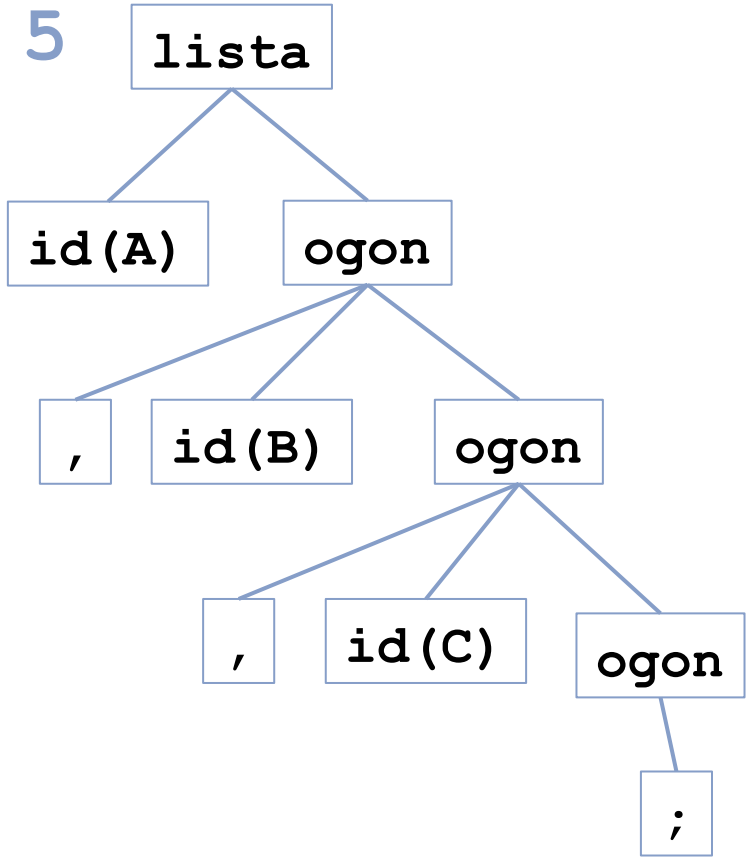
- Wejście (ciąg napisów)

A , B , C ;

- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

- 5



Parsery z rekurencją zstępującą 1

- Każdemu symbolowi nieterminalnemu gramatyki odpowiada podprogram, który przegląda zdania generowane przez ten symbol
- Opis EBNF gramatyk jest wygodny przy budowie parserów stosujących rekurencję zstępującą gdyż ogranicza liczbę symboli nieterminalnych.

Parsery z rekurencją zstępującą 2

- Gramatyka prostego wyrażenia arytmetycznego (EBNF)

```
expr   →  term  {(+ | -) term }
term   →  factor  {(* | /) factor }
factor  → id | int_constant | ( expr )
```

Parsery z rekurencją zstępującą 3

- Kod
  - Zakładamy, że skaner o nazwie lex() umieszcza kod następnego tokenu w zmiennej **`kolejnyToken`**
  - Proces kodowania w przypadku pojedynczej prawej strony wygląda następująco:
    - porównaj każdy T po prawej stronie reguły z następnym tokenem; jeśli są jednakowe kontynuuj, w przeciwnym razie zgłaszaj błąd
    - dla każdego N po prawej stronie reguły wywołaj odpowiadającą symbolowi funkcję parsowania

Parsery z rekurencją zstępującą 4


```
/* Function expr
   Sprawdza napisy w języku generowanym regułą:
    expr  ->  term  {(+ | -)  term }
 */

void expr() {

   /* Parsuj term */

  term();

  /* dopóki następny token to + lub - dopóty wywołuj
     lex by wczytać kolejny token i parsuj nastepny term */

   while (kolejnyToken == ADD_OP || kolejnyToken == SUB_OP){
        lex();
        term();
   }
}
```

```
expr   →  term  {(+ | -)  term }
term   →  factor  {(* | /) factor }
factor  → id | int_constant | ( expr )
```

Parsery z rekurencją zstępującą 5


```
/* term
   Sprawdza napisy w języku generowanym regułą:
    term  ->  factor  {(* | /)  factor )
*/
void term() {
        printf("Enter term \n");
        /* Parsuj pierwszy factor */
        factor();

        /* dopóki następny token to * lub / dopóty wywołuj
           lex by wczytać następny token i parsuj następny factor *
        while (kolejnyToken == MULT_OP) || kolejnyToken == DIV_OP) {
                lex();
                factor();
        }
        printf("Exit term \n");
} /* Koniec funkcji term */
```

```
expr   →  term  {(+ | -)  term }
term   →  factor  {(* | /) factor }
factor  → id | int_constant | ( expr )
```

Parsery z rekurencją zstępującą 6


```
/* factor
   Sprawdza napisy w języku generowanym
   regułą:
   factor  -> id | int_constant | ( expr )
*/
void factor() {
   printf("Enter  factor \n");
/* Wyznacz RHS */
if (kolejnyToken == IDENT ||
    kolejnyToken == INT_LITERAL){
    /* Pobierz następny token */
    lex();
/* Jeśli prawa strona to (expr), wywołaj
   lex by minąć lewy nawias (, wywołaj
   expr() i sprawdź prawy nawias ). */
else {
   if (kolejnyToken == LEFT_PAREN) {
      lex();
      expr();
   if (kolejnyToken == RIGHT_PAREN)
      lex();
   else
      error();
   } /*koniec
       if(kolejnyToken==LEFT_PAREN */
   /* To nie jest id ani )  /*
   else
      error();
   } /* Koniec else */
   printf(”Exit z factor\n");;
} /* Koniec funkcji factor */
```

```
expr   →  term  {(+ | -)  term }
term   →  factor  {(* | /) factor }
factor  → id | int_constant | ( expr )
```

Parsery z rekurencją zstępującą 7

- Umowa:

  każda funkcja umieszcza następny token
  w zmiennej **`kolejnyToken`**

Parsery z rekurencją zstępującą 8

- Ślad analizatora leksykalnego i parsera na wyrażeniu

```
(sum + 47) / total
```

```
Next token is: 25 Next lexeme is (
Enter expr
Enter term
Enter factor
Next token is: 11 Next lexeme is
   sum
Enter expr
Enter term
Enter factor
Next token is: 21 Next lexeme is +
Exit factor
Exit term
Next token is: 10 Next lexeme is 47
Enter term
Enter factor
Next token is: 26 Next lexeme is )
Exit factor
Exit term
Exit expr
Next token is: 24 Next lexeme is /
Exit factor
Next token is: 11 Next lexeme is total
Enter factor
Next token is: -1 Next lexeme is EOF
Exit factor
Exit term
Exit expr
```

Parsowanie `function`

- Weźmy następującą gramatykę:

```
program        =>  lista_funkcji
lista funkcji  =>  lista_funkcji funkcja | funkcja
funkcja        =>  FUNC nazwa(lista_param) instrukcje
```

- Odpowiadająca funkcja parsera dla s. “funkcja”:

```
void parsujFunkcja() {
     sprawdzajToken(T_FUNC);
     parsujIdent();
     sprawdzajToken(L_PAREN)();
     parsujListaParam();
     sprawdzajToken(R_PAREN);
     parsujInstrukcje();
}
```

Parsowanie `function`

- funkcja **`sprawdzajToken()`**

```
void sprawdzajToken(int spodziewany) {
    if (kolejnyToken != spodziewany) {
    printf("błąd składni, spodziewany %d, kolejny%d\n",
spodziewany, kolejnyToken);
    exit(0);
} else  // idź dalej
    kolejnyToken == yylex();
}
```

Problem rekurencji lewostronnej

- Gramatyka (poprzednia)

```
lista_funkcji => lista_funkcji funkcja | funkcja
funkcja  => FUNC Ident ( lista_param ) Instrukcje
```

... prowadzi do niekończących się pętli w funkcjach parsera. Należy ją przedefiniować. Np.

```
void parsujListaFunkcji() {
   parsujListaFunkcji(); // rekurencja! Źle!
   parsujFunkcja();
}
```

Parsery z rekurencją zstępującą 9

- Klasa gramatyk LL
  - Problem rekurencji lewostronnej (np. $A \rightarrow A + B$)
    - Parsery top-down nie mogą pracować w oparciu o gramatyki z rekurencją lewostronną
    - W wypadku rekurencji bezpośredniej gramatykę można przekształcić. Dla każdego symbolu nieterminalnego A:
      - Zgrupować reguły następująco:

        $$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

        gdzie żadne β nie zaczyna się od A;
      - Oryginalne reguły zastąpić przez:

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

Przykłady eliminacji l-rekurencji

- Zastosujemy opisaną procedurę do następującej gramatyki:
  - E → E + T T
  - T → T * F F
  - F → (E) id
- Dla E-reguł mamy α_1 = + T i β = T, więc możemy zastąpić podaną regułę dla E przez
  - E → T E'
  - E → + T E' | ε
- Dla reguł T mamy α_1 = * F i β = F, więc zastępujemy je przez
  - T→ F T'
  - T → * F T' | ε
- Ponieważ nie ma lewostronnej rekurencji w regułach F to pozostają. Nowa gramatyka jest:
  - E → T E
  - E → + T E
  - T→ F T
  - T → * F T
  - F → (E) id
- Gramatyka ta generuje ten sam język co poprzednia i nie jest l-rekurencyjna.

Parsowanie funkcji cd.

- Wracamy do gramatyki “funkcji”
- Przepisujemy

```
lista_funkcji => lista_funkcji funkcja | funkcja
```

do postaci

```
lista_funkcji => funkcja lista_funkcji | funkcja
```

i faktoryzujemy:

```
lista_funkcji => funkcja wiele_funkcji
wiele_funkcji => funkcja wiele_funkcji | ε
```

Poprawiona procedura parsera:

```
void parsujListaFunkcji() {
   parsujFunkcja();     // Dobrze.
   parsujWieleFunkcji();
}
```

Parsowanie instrukcji `if`

- Instrukcja `IF`

```
if_stat => IF expr THEN stat ENDIF
         | IF expr THEN stat ELSE stat ENDIF
```

Dla parsera z rekurencją zstępującą podana gramatyka prowadzi do błędów. Należy ją przekształcić. Faktoryzujemy:

```
if_stat => IF expr THEN stat closeif
closeif => ENDIF | ELSE stat ENDIF
```

Parsowanie instrukcji `if`

- Procedura parsera

```
void parsujStat() {
        sprawdzajToken(T_IF);
        parsujExpr();
        sprawdzajToken(T_THEN);
        parsujStat();
        parsujCloseIf();
}
void parsuj CloseIf() {
        if (kolejnyToken == T_ENDIF)
                kolejnyToken = yylex();
        else {
                sprawdzajToken(T_ELSE);
                parsujStat();
                sprawdzajToken(T_ENDIF);
} // najpierw sprawdzamy ENDIF i jeśli się nie zgadza sprawdzamy ELSE
```

Wielokrotna produkcja - **First**

- Jak to się robi gdy mamy więcej produkcji, np.

```
stat => assign_stat | return_stat | print_stat
          | empty_stat | if_stat | while_stat
          | block_stat
```

Jak napisać parsujStat() by odróżnić wszystkie możliwości? (Pamiętamy, że mamy jeden symbol z wejścia i parser pracuje metodą rekurencji zstępującej.)

Wprowadzimy pojęcie zbiorów **First**.

Def. Zbiorem **First**(α) dla symbolu α jest zbiór terminali, które można wyprowadzić z α jako pierwsze. Dokładniej: rozważamy wszystkie napisy wyprowadzane z α . Jeżeli α =>*β i następnie β=>aγ to terminal a$\in$**First**(α).

Dalej podamy algorytm znajdowania **FIrst**().

Zastosowanie **First**

- Jeżeli produkcja ma wiele możliwych prawych stron to zbiory FIRST() pozwalają rozstrzygać, którą możliwość wybrać ma parser.

Schemat procedury parsera dla produkcji $A \Rightarrow \alpha_1 | \alpha_2 | \ldots$:

```
void parsujA() {
   switch (kolejnyToken) {
      case FIRST(α1) :
         /*     kod rozpoznający α1     */
        return;
      case FIRST(α2) :
         /*     kod rozpoznający α2     */
        return;
        ...
       default:
          printf("błąd \n");
          EXIT(0);
   }
}
```

Wielokrotna produkcja - **Follow**

- Jeśli symbol nieterminalny zeruje się wówczas postępujemy inaczej. Symbol ten znika w parsowanym napisie (ε ∈ **First**). Symbol A można więc opuścić i następny token będzie pierwszym tokenem symbolu, który stoi po A w parsowanym napisie. Parser powinien rozpoznawać sytuacje, w których A =>* ε.

- Def. Zbiór **Follow**(A) symbolu nieterminalnego A jest zbiorem symboli terminalnych, które mogą występować po A. Dokładniej: dla każdego poprawnego zdania S =>* αAβ gdzie β rozpoczyna się symbolem terminalnym b, b∈**Follow**(A). Dalej podany jest algorytm znajdowania **Follow**().

Znajdowanie **First** i **Follow**

- **First**(α)

α=$X_1X_2...X_n$, Z=zerowalne

1. Jeśli X_1 jest T, dodaj go
2. Jeśli X_1 nie jest T, dodaj **First**(X1)-ε
   - Jeśli X_1 jest zerowalny, dodaj FIRST(X_2)-ε . Jeśli X_2 jest zerowalny, dodaj FIRST(X_3)-ε itd. aż do X_n.
   - Jeśli α =>* ε dodaj ε.

- **Follow**(A)

1. Umieść $ w **Follow**(S), S=symb. startowy, $ =koniec wejścia;
2. ⋀A => αBβ gdzie B nie jest terminalny, dodaj **First**(β) z wyjątkiem ε do **Follow**(B)
3. ⋀A => αB | αBβ gdzie **First**(β) zawiera ε (zerowalne β) dodaj do **Follow**(B) zbiór **Follow**(β).

Przykład **First**

- Gramatyka
  - S => AB
    A => Ca | ε
    B => BaAC | c
    C => b | ε

    Ponieważ gramatyka jest lewostronnie rekurencyjna, poprawiamy ją.
  - S => AB
    A => Ca | ε
    B => cB'
    B' => aACB' | ε
    C => b | ε
    (B=> BaAC | c; zamianiamy na B=>cB', B'=>aACB' | ε)

- **First**(C) = {b ε}
- **First**(B') = {a ε}
- **First**(B) = {c}
- **First**(A) = {b a ε}
  (zaczynamy z FIRST(C)-ε, dodajemy a, bo C jest zerowalny i dodajemy ε, bo A jest zerowalny)
- **First**(S) = {b a c}
  (zaczynamy z **First**(A)-ε, dodajemy **First**(B),, bo S nie jest samozerowalny – A może znika lecz B zostaje)

Przykład **Follow**

- Dla tej samej gramatyki:
  - S => AB
    A => Ca | ε
    B => cB’
    B’ => aACB’ | ε
    C => B | ε

  wyliczamy zbiory **Follow**
- Dla każdego symbolu nieterminalnego przechodzimy po prawych stronach produkcji, w których symbol występuje i wykonujemy algorytm podany wcześniej.

- **Follow**(S) = {$}
  $ nie występuje z lewej, ale S jest symb. startowym i dodajemy $ zgodnie z regułą.
- **Follow**(B) = {$}
  B jest z prawej strony S=>AB. **Follow**(B) jest taki sam jak dla S.
- **Follow**(B’) = {$}
  Dwie prawe strony zawierają B’: B’=>aACB’ – ta nie daje nic i B=>cB’ – z tej reguły wnosimy, że B’ ma ten sam **Follow** co B.
- **Follow**(C) = (a $)
- **Follow**(A) = {a b c $}
  Z S=AB wnosimy, że należy dodać **First**(B)={ε}. Z B’=aACB’ dodajemy **First**(C)={b}. Ponieważ C jest w Zero(G) to włączamy też **First**(B’)={a}; B’ jest w Zero(G) więc dodajemy **Follow**(B’)={$}.

Wykorzystanie **Follow**

- Możemy teraz uogólnić procedurę parsoawnia

```
void parsujA() {
   switch (kolejnyToken) {
      case FIRST(α1) :
         /*     kod rozpoznający α1     */
        return;
      case FIRST(α2) :
         /*     kod rozpoznający α2     */
        return;
         ...
       case FOLLOW(A): // A=>epsilon
          /* zazwyczaj nic nie robimy  */

       default:
          printf("błąd \n");
          EXIT(0);
   }
}
```

Parsery z rekurencją zstępującą 10

- Własnością gramatyk, która nie pozwala na typ parsowania z rekurencją zstępującą jest brak rozłączności reguł gramatycznych
  - nie można jednoznacznie wybrać reguły na podstawie następnego wczytanego tokenu
  - Definicja:

$$\textbf{First}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\} \quad (\text{If } \alpha \Rightarrow^* \varepsilon,\ \varepsilon \in \textbf{First}(\alpha))$$

W ostatnim przypadku symbol α jest zerowalny - może prowadzić do pustej prawej strony produkcji.

Parsery z rekurencją zstępującą 11

- Test rozłączności par:
  - dla każdego A, które posiada więcej prawych stron niż jedna i dla każdej pary reguł

$$A \rightarrow \alpha_i, \quad A \rightarrow \alpha_j$$

musi zachodzić

$$\mathbf{First}(\alpha_i) \cap \mathbf{First}(\alpha_j) = \varnothing$$

Przykłady

- Przykład 1

  A → aB | bAb | Bb

  B → cB | d

Zbiory **First**: {a}, {b}, {c,d} są parami rozłączne

- Przykład 2

  A → a | bB | cAb

  A → a | aB

Zbiory **First**: {a,b,c} ∩ {a} ≠ ∅

- Przykład 3

  A aB | BAb

  B aB | b

Zbiory **First** są: {a} i {a,b}. Patrząc na pierwszy symbol z wejścia, program nie jest w stanie rozstrzygnąć, która reguła powinna być użyta.

Parsowanie wspomagane tablicą

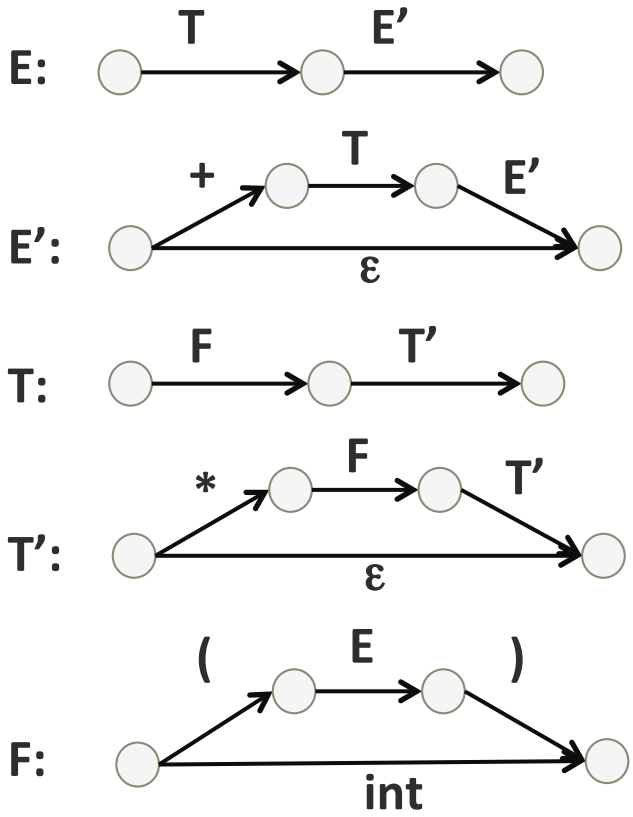
- Pomocnicza tablica zawiera produkcje i razem z zawartością stosu pozwala określić gdzie znajduje sią parser.
- Gramatyka (proste wyrażenia wraz z kolejnością działań i łącznością):
- E => E+T | T
  T => T*F | F
  F => (E) | int
- Po eliminacji rekurencji:
- E => TE’
  E’ => +TE’ | ε
  T => FT’
  T’ => *FT’ | ε
  F => (E) | int

Parsowanie z tablicą

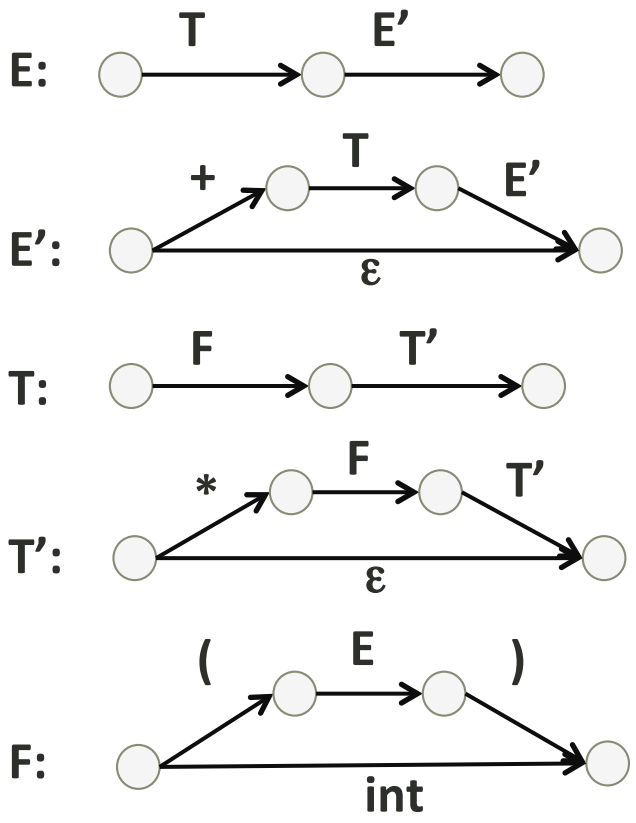
GRAMATYKA

- E => TE’
  E’ => +TE’ | ε
  T => FT’
  T’ => *FT’ | ε
  F => (E) | int

- Diagram pomocniczy
  (stany + funkcje przejścia)



Parsowanie z tablicą



- Analiza: **`int+int*int`**

Zaczynamy od stanu pierwszego w E. Funkcja T prowadzi do następnego stanu. Zostawiamy to i idziemy do diagr. T. Tu zmianę stanu wyznacza F. W F mamy terminale. Czytamy token z wejścia. Musi to być albo (albo int. Mamy int, więc opuszczamy ją i wracamy do 2go stanu T. Mamy teraz T’. Idziemy tam. Tu widzimy *. Z wejścia mamy +. Nie ma dopasowania, ale jest w T’ produkcja zerowa, więc pomijamy T’ i wracamy do T i ponieważ funkcją przejścia jest T’ więc pomijamy to i wracamy do 2go stanu w E gdzie zaczynaliśmy. Przechodzimy do E’ i powtarzamy aż do wyczerpania wejścia.

Parsowanie z tablicą

- Parser sterowany tablicą używa stosu do pamiętania produkcji, do których musi wracać. Tablica parsera zawiera akcje, które parser powinien wykonać opierając się na tokenie z wejścia i wartości na szczycie stosu.

Tablica M parsera

- Tablica M dla gramatyki:

  - **E => TE’**
    E’ => +TE’|ε
    T => FT’
    T’ => *FT’|ε
    F => (E) | int

wejście: stos	`int`	+	*	(	)	$
E	E->TE’			E->TE’		
E’		E’->+TE’			E’->ε	E’->ε
T	T->FT’			T->FT’		
T’		T’->ε	T’->*FT’		T’->ε	T’->ε
F	F->int			F->(E)		

Na stos wkładamy symbol startowy (S) i wczytujemy pierwszy token z wejścia. Możliwe sytuacje jeśli na szczycie stosu znajduje się symbol **X**, mamy **a** z wejścia i tablicę **M**:

1. X == a == $: Parser zatrzymuje się. **Accept**.
2. X == a != $, jest dopasowanie: pop X, weź następny token z we. **Match**.
3. X != a i X jest nieterminalny: pop X, zastosuj produkcję M[X,a], włóż prawą stronę produkcji na stos. **Predict**.
4. Jeśli żadna z sytuacji nie wystąpiła lub M[,] w kroku 3. jest pusta. **Error**.

Tablica M parsera - inaczej

- Tablica M dla gramatyki

  - **1 E => TE'**
    2, 3 E' => +TE'|ε
    4, T => FT'
    5, 6 T' => +FT'|ε

    7, 8 F => (E) | int

    (ponumerowano produkcje)

wejście: stos	`int`	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

Przykład

1 E => TE'
2, 3 E' => +TE' | ε
4 T => FT'
5, 6 T' => *FT' | ε
7, 8 F => (E) | int

Ślad parsowania (int+int)

STOS	Reszta WE	Akcja parsera
E$	(int + int)	Predict. E=>TE' pop E, push TE', z WE:ϕ
TE'$	(int + int)	Predict, T=>FT'
FT'E'$	(int + int)	Predict, F=>(E)
(E)T'E'$	(int + int)	Match (, pop (, WE:int
E)T'E'$	int + int)	Predict, E=>TE'
TE')T'E'$	int + int)	Predict, T=>FT'
FT'E')T'E'$	int + int)	Match, pop int, WE:+
T'E')T'E'$	+ int)	Predict T'=>ε
E')T'E'$	+ int)	Predict, E'=>+TE'
+TE')T'E'$	+ int)	Match +, pop +, WE:int
TE')T'E'$	int)	Predict, T=>FT'
FT'E')T'E'$	int)	Predict F=>int
intT'E')T'E'$	int)	Match int, pop int, WE:)
T'E')T'E'$	)	Predict, T'=>ε
E')T'E'$	)	Predict E'=>ε
)T'E'$	)	Match), pop), WE:$
T'E'$	$	Predict, T'=>ε
E'$	$	Predict, E'=>ε
$	$	Match $, pop), Success.

TABLICA PARSERA M(X,a)

we: stos	int	+	*	(	)	$
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

Jak zbudować tablicę M parsera?

(raczej komputerowo; $ = koniec danych z wejścia)

1. Znaleźć zbiory **First** i **Follow**
2. Wiersze M[,] numerujemy symbolami nieterminalnymi G, kolumny symbolami terminalnymi
3. Algorytm
   1. Dla każdej produkcji A=>α wykonaj kroki a) i b):
      a) dla każdego symbolu terminalnego **a**$\in$**First**(α) dodaj A=>α do M[A,a]
      b) Jeśli $\varepsilon \in$ **First**(α) (A jest zerowalne) dodaj A=>α do M[A,b] dla każdego **b**$\in$**Follow**(A). Jeśli $\varepsilon \in$ **First**(α) i \$$\in$**Follow**(A) dodaj A=>$\alpha$ do M[A,\$]
   2. Puste M[,] oznaczają błędy.

Parsery typu bottom-up

LR

Parsery typu bottom-up 1

- Zadaniem parsera typu *bottom-up* jest znalezienie właściwej prawej strony (PSR) w redukowanej formie zdaniowej tak by otrzymać poprzedzającą formę zdaniową w produkcjach

- PSR = prawa strona reguły gramatycznej;
- LSR = lewa strona reguły gramatycznej;

Przykład

- GRAMATYKA

1. E → E + T | T
2. T → T * F | F
3. F → (E) | id

- PRZYKŁAD PRODUKCJI

E → E + T
→ E + T * F
→ E + T * id
→ E + F * id
→ E + id * id
→ T + id * id
→ F + id * id
→ id + id * id

Podkreślone części w każdej produkcji są prawymi stronami tych form zdaniowych, które zastępowane są przez odpowiadające im lewe strony w celu otrzymania poprzedzającej formy zdaniowej. Parser bottom-up rozpoczyna od wyrażenia na dole (wczytane zdanie) i produkuje ciąg form zdaniowych do momentu aż ostatnią formą jest symbol startowy (tutaj E). W każdym kroku zadaniem parsera jest znalezienie takiej prawej strony - uchwytu - we wzorcu (formie), który musi po zastąpieniu dać następną (poprzednia) formę. Np. E + T * id zawiera trzy prawe strony E+T, T i id. Tylko jedna z nich stanowi uchwyt. Jeśli np. weźmiemy jako prawą stronę do zamiany E+T to wynikowa forma będzie postaci E*id, ale E*id nie jest dopuszczalna dla tej gramatyki ...

Przykład (d-g) - 1

- Wejście (ciąg napisów)

A , B , C ;

- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Parser

- CZYTA I SKŁADA NA STOSIE WSZYSTKIE LEKSEMY AŻ DO MOMENTU GDY STWIERDZI, ŻE ELEMENTY STANOWIĄ KOMPLETNĄ PRAWĄ STRONE JEDNEJ Z PRODUKCJI (w tym przykładzie całe wejście)

STOS po każdym kolejnym leksemie z wejścia:

- **id(A)**
- **id(A) ,**
- **id(A) , id(B)**
- **id(A) , id(B) ,**
- **id(A) , id(B) , id(C)**
- **id(A) , id (B), id(C) ;**

W tym momencie można zidentyfikować średnik **;** jako **ogon** (reguła 3).

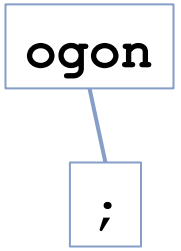
Przykład (d-g) - 2

- Wejście (ciąg napisów)

A , B , C ;

- Gramatyka

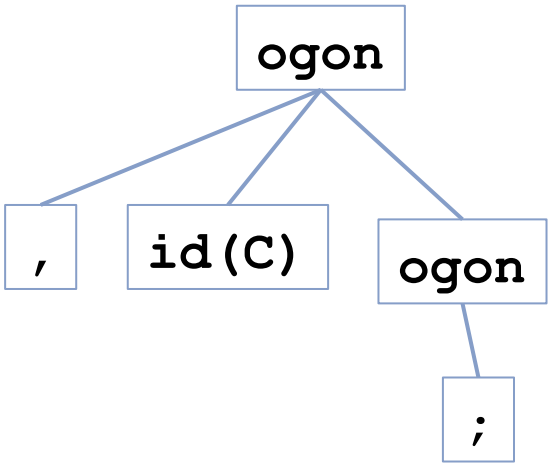
```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```



Przykład (d-g) - 3

- Wejście (ciąg napisów)

A , B , C ;



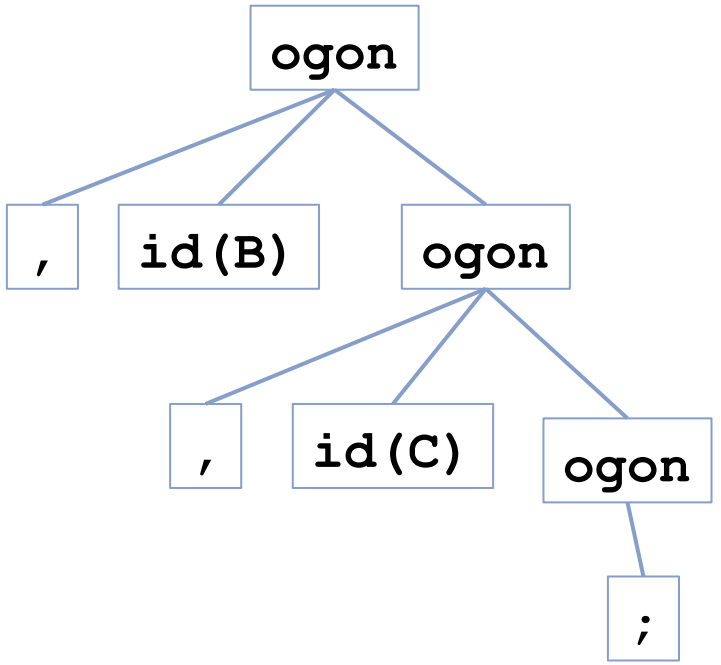
- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Przykład (d-g) - 4

- Wejście (ciąg napisów)

A , B , C ;



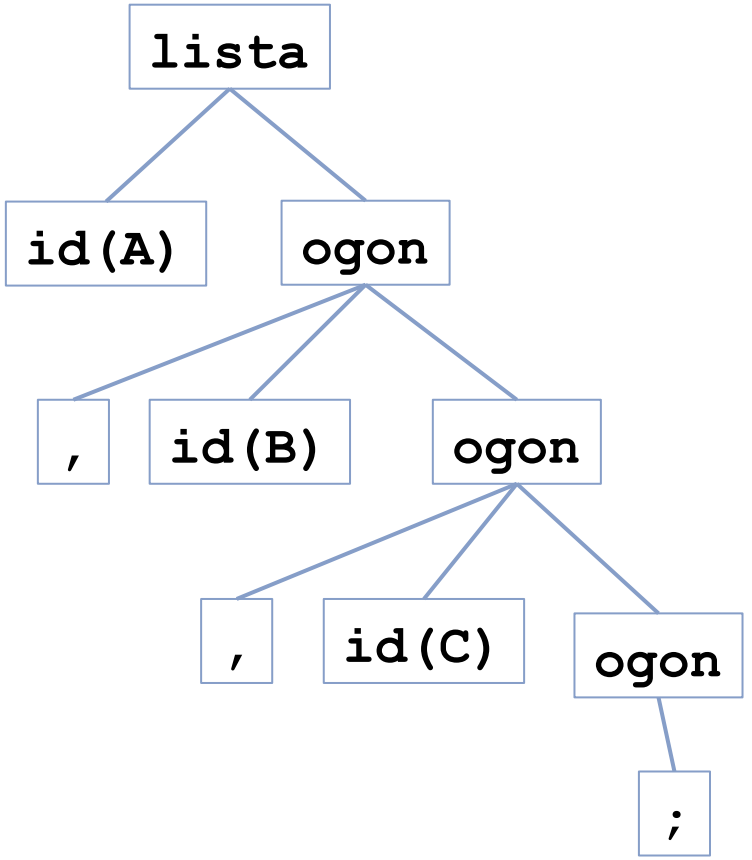
- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Przykład (d-g) - 5

- Wejście (ciąg napisów)

A , B , C ;



- Gramatyka

```
lista => id ogon
ogon  => , id ogon
ogon  => ;
```

Parsery typu bottom-up 2

- Pojęcie “uchwytu” (definicje)
  - β jest uchwytem formy zdaniowej γ:
    $$\gamma = \alpha\beta w \Leftrightarrow S =>^*_r \alpha A w =>_r \alpha\beta w$$
  - β jest frazą formy zdaniowej γ:
    $$\gamma \Leftrightarrow S =>^* \gamma = \alpha_1 A \alpha_2 =>^+ \alpha_1\beta\alpha_2$$
  - β jest prostą frazą zdania γ:
    $$\gamma \Leftrightarrow S =>^* \gamma = \alpha_1 A \alpha_2 => \alpha_1\beta\alpha_2$$

Parsery typu bottom-up 3

- Pojęcie “uchwytu” cd.
  - Uchwytem formy zdaniowej jest pierwsza lewostronna fraza prosta
  - Łatwo wskazać uchwyt w przypadku gdy istnieje drzewo parsowania

Parsery typu bottom-up 4

- Algorytmy typu shift-reduce
  - Reduce jest akcją zamiany uchwytu na szczycie stosu parsera przez odpowiadającą mu lewą stronę reguły gramatycznej
  - Shift oznacza akcję umieszczenia następnego tokenu na stosie parsera

Parsery typu bottom-up 5

- Korzyści używania parserów LR:
  - Pracują prawie dla wszystkich typów gramatyk języków programowania.
  - Są tak samo wydajne jak inne parsery typu bottom-up
  - Wyłapują błędy najszybciej jak można.
  - Klasa gramatyk LR jest nadzbiorem klasy przeglądanej przez parsery LL.

Parsery typu bottom-up 6

- Parsery b-u budowane są z pomocą programów narzędziowych
- Cała historia przeglądania wejścia może być potrzebna do podjęcia decyzji w danym miejscu (Knuth, D., twórca parserów LR); w praktyce jest lepiej
- Konfiguracja LR informuje o aktualnym stanie parsera LR

$(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$

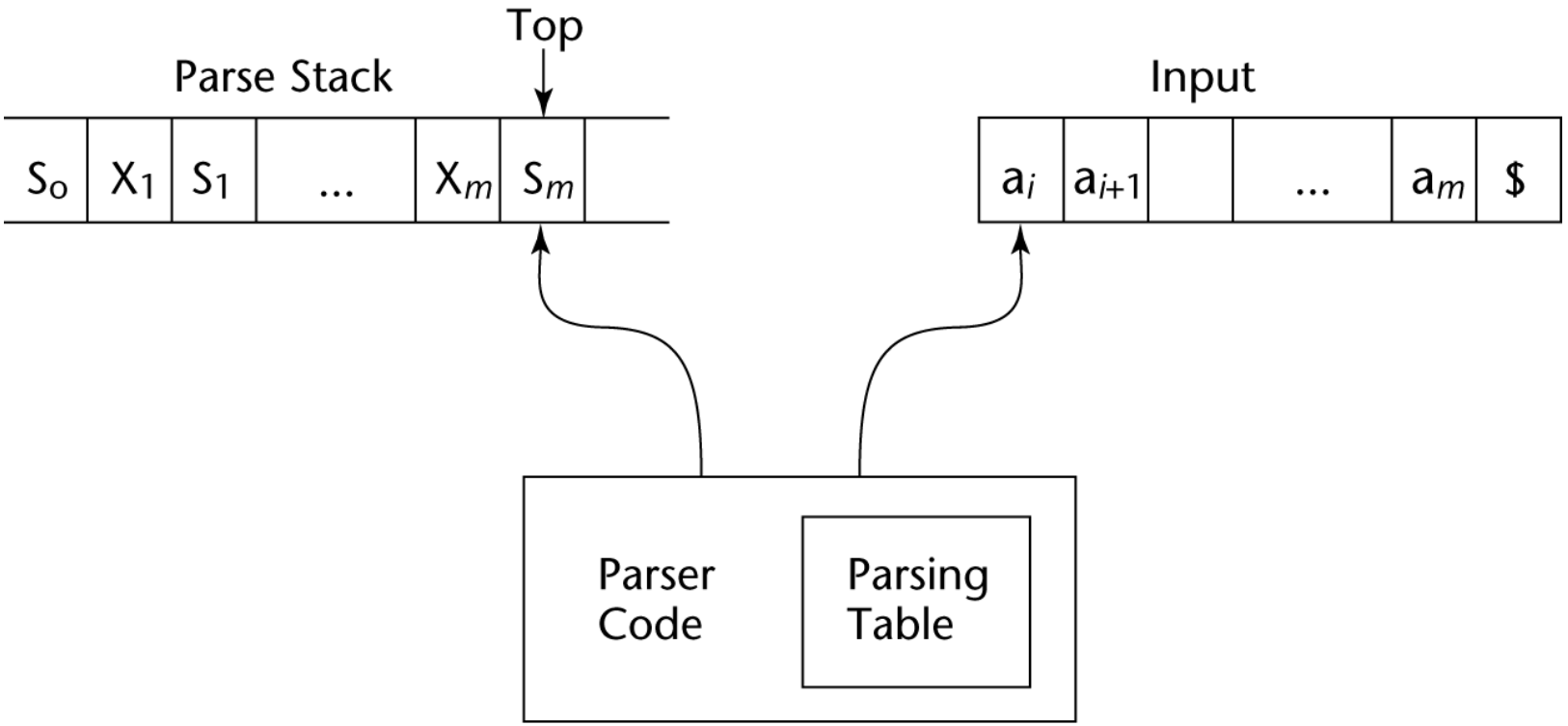
Donald Knuth (ur. 1938)

Wybitny matematyk, informatyk, twórca systemów TeX i Metafont. Autor cyklu książek “Sztuka programowania”, “Matematyka dyskretna”. Algorytmy z dziedziny informatyki... (profesor emeryt, Stanford University, USA)

Parsery typu bottom-up 7

- Parsery LR sterowane są dwiema tablicami: **Action** i Goto
  - Tablica Action określa akcje parsera na podstawie jego stanu i następnego tokenu
    - Wiersze są stanami, a kolumny s. terminalnymi
  - Tablica Goto określa stan parsera, który należy umieścić na stosie po redukcji
    - Wiersze są nazwami stanów; kolumny są symbolami nieterminalnymi

Struktura parsera LR



Parsery typu bottom-up 8

- Konfiguracja początkowa: : $(S_0, a_1 \ldots a_n\$)$
- Akcje parsera
  - Shift: umieszczenie na stosie następnego symbolu z wejścia jednocześnie z symbolem stanu, będącym częścią specyfikacji Shift z tablicy **Action**
  - Reduce: usunięcie uchwytu ze stosu wraz z jego symbolem stanu. Umieszczenie na stosie LSR. Umieszczenie na stosie symbolu z tablicy **Goto** z użyciem symbolu stanu leżącym tuż poniżej nowej LSR na stosie i nowej LSR zgodnie z wierszem i kolumną tablicy **Goto**.

Parsery typu bottom-up 9

- Akcje parsera (cd.)
  - Accept: OK. Ukończono parsowanie. Nie wykryto błędów.
  - Error: parser wywołuje procedurę błędów

Przykład

Gramatyka wyrażeń arytmetycznych

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Następna strona pokazuje tablicę parsowania LR dla gramatyki pokazanej w lewej kolumnie. R oznacza reduce, S shift. R2 oznacza reduce wg. reguły 2; S7 oznacza shift wg. reguły 7 (włóż stan S7 na stos). Puste kratki w tab. **Action** wskazują na błąd syntaktyczny (można tu wywołać odpowiednie procedury obsługi błędów). Tablice parsowania można otrzymać narzędziem **`yacc`** lub podobnym.

Tablica parsera LR

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Parsery typu bottom-up 10

- Tablicę parsera LR można wygenerować z pomocą programów narzędziowych, np. **`yacc`** lub **`bison`**

Podsumowanie i dyskusja

!?

Podsumowanie

- Analiza syntaktyczna stanowi część implementacji każdego języka programowania
- Analizator leksykalny
  - wydziela najmniejsze składniki programu
  - znajduje błędy
  - tworzy drzewo składni (parsowania)
- Parsery używające rekurencji zstępującej – LL
  - oparte na gramatykach w notacji Backusa-Naura (EBNF)
- Zadanie parserów bottom-up: znaleźć podnapis bieżącej formy zdaniowej
- Najszerzej używany typ parserów to parsery LR, stosujące technikę shift-reduce; bottom-up

Dodatki

First/Follow/Predict

- Algorytmy **First/Follow/Predict**:

  - **First**(α) == {a : $\alpha \rightarrow^*$ a β}
    $\cup$ (**if** α =>* ε **then** {ε} **else** ϕ)
    = (zbiór tokenów początkowych A)

  - **Follow**(A) == {a : S $\rightarrow^+ \alpha$ A a β}
    $\cup$ (**if** S $\rightarrow^* \alpha$ A **then** {ε} **else** ϕ)
    = (zbiór tokenów, które następuja po A)

  - **Predict** (A $\rightarrow$ X_1 ... X_m) == (**First** (X_1 ... X_m) - {ε})
    $\cup$ (**if** X_1, ..., X_m $\rightarrow^*$ ε **then Follow** (A) **else** ϕ)

Przykład (wp)

Gramatyka

(0) Z → E

(1) E → E * B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

Jeśli produkcja jest postaci A→ αβ to sytuację oznacza się jako [A→α•β]. Np produkcja E→E+B ma cztery sytuacje E→•E+B, E→E•+B, E→E+•B, E→E+B•

Sytuacje i przejścia między nimi są podstawą budowy parserów.

Zbiór sytuacji

{

1. {B → 0 •},
2. {B → 1 •},
3. {Z → E •, E → E • * B, E → E • + B},
4. {E → B •}, {E → E * • B + B → • 0, + B → • 1},
5. {E → E + • B, + B → • 0, + B → • 1},
6. {E → E * B •},
7. {E → E + B •}

}

(Zbiory domknięte)

Słowniczek

- **Diagram stanów.** Graf skierowany. Węzłami są nazwy stanów. Przy łukach diagramu umieszcza się znaki z wejścia, które generują przejścia między stanami, jak również akcje skanera
- **Automat skończony.** Równoważnik matematyczny diagramu stanów. Graf jest reprezentacją graficzną automatu.
- **Język regularny**. Język rozpoznawany (generowany) przez automat skończony.
- **LL** = Left to right, Leftmost first, **LR** = Left to right, Rightmost first, **LL(1)** = **LL** + jeden token "do przodu".
- ⇔ wtedy i tylko wtedy.
- **T** - symbol terminalny (a, b, c); **N** – symbol nieterminalny (A, B, C).
- **LSR** = lewa strona reguły, PSR = prawa strona reguły (gramatycznej)
- $\Rightarrow_{rm}$ krok produkcji *rightmost*, $\Rightarrow^*_{rm}$ - zero lub więcej kroków produkcji *rm.*
- l-rekurencyjna = lewostronnie rekurencyjna; podobnie p-rekurencyjna...

Słowniczek

- **Forma zdaniowa** = wyprowadzenie z symbolu początkowego; napis α należący do (T ∪ N)* taki, że S =>* α. Mozemy mieć lewostronne lub prawostronne formy zdaniowe.

Symbole

- $\varnothing$ – zbiór pusty
- ε – pusta instrukcja
- $\cup$ – suma mnogościowa
- $\in$ – należy
- $\notin$ – nie należy
- $ – koniec danych

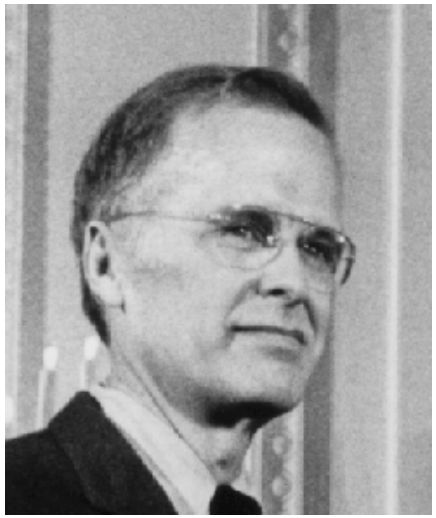
Zadania

- G jest gramatyką:
  - G → S $
    S → A M
    M → S | ε
    A → a E | b A A
    E → a B | b A | ε
    B → b E | a B B
  - Opisać słownie generowany język. Dla łańcucha abaa znaleźć drzewo parsowania. Czy jest to gramatyka LL(1)? Jeśli tak wypisać tablicę parsera. Jeśli nie podać konflikty.
- Podać schemat programu znajdowania zbiorów **First**(A), **Follow**(A), **Predict**(A=>α).
-

Kto, co?

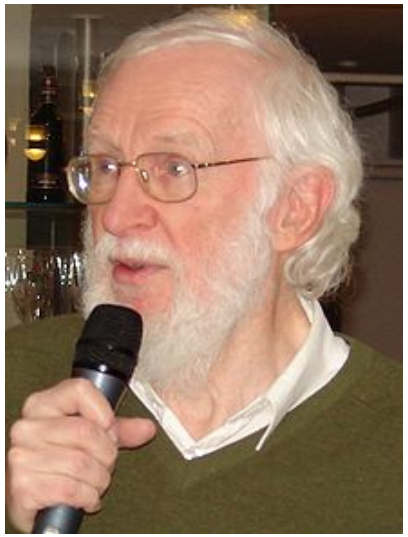
BNF

Gramatyki



John Backus
(1924-2007)

Peter Naur (1928)



Noam Chomsky
(1928)

za tydzień ... !?