Przetwarzanie tekstu 8-9-10-11

Analiza składniowa

Gramatyki bezkontekstowe

definicja

Gramatyką bezkontekstową nazywamy czwórkę

$$\langle N, T, P, S \rangle$$

gdzie

- N - zbiór symboli nieterminalnych
- T - zbiór symboli terminalnych
- $P \subset N \times (T \cup N)^*$ - zbiór produkcji
- $S \in N$ - symbol początkowy gramatyki

Gramatyki bezkontekstowe

definicja

symbole terminalne (terminale) to symbole występujące w definiowanym języku (= elementy alfabetu języka)

symbole nieterminalne (nieterminale) to symbole pomocnicze, wprowadzane dla nazwania typów podwyrażeń języka (takich, jak np. instrukcja, lista argumentów, wyrażenie, blok, program; fraza rzeczownikowa, zdanie względne).

produkcje są regułami budowania większych *wyrażeń* z mniejszych. Produkcję $\langle A, \alpha \rangle$ zapisuje się $A \rightarrow \alpha$

Gramatyki bezkontekstowe

relacja wywodzenia

- produkcje to reguły przepisywania
- pomiędzy ciągami symboli z $(N \cup T)^*$ definiujemy relację bezpośredniego wywodzenia $\Rightarrow$:

$$\alpha A \beta \Rightarrow \alpha \gamma \beta \qquad \text{jeśli} \qquad A \to \gamma \in P$$

- zwrotne i przechodnie domknięcie relacji bezpośredniego wywodzenia nazywamy relacją wywodzenia i oznaczamy $\stackrel{*}{\Rightarrow}$

$$\alpha_0 \stackrel{*}{\Rightarrow} \alpha_n \quad \equiv \quad \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n , \quad n \geq 0$$

- ciąg bezpośrednich wyprowadzeń takich, że lewa strona kolejnego jest prawą stroną poprzedniego nazywamy wywodem.

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$$

- ciąg symboli $\alpha \in (N \cup T)^*$, który da się wywieść z symbolu początkowego gramatyki, czyli taki że $S \stackrel{*}{\Rightarrow} \alpha$, nazywamy formą zdaniową.

Gramatyki bezkontekstowe

przykład

- gramatyka

  $G = \langle \{ S, A, B \}, \{ \mathrm{a}, \mathrm{b} \}, P, S \rangle$

$$P: \quad \begin{array}{lll} \underline{S} \to \mathrm{a}\ B & (1) \\ \underline{S} \to B & (2) \\ B \to \mathrm{b} & (3) \\ B \to \mathrm{bb} & (4) \end{array}$$

- wywód słowa a b b z S

  $\underline{S} \underset{1}{\Rightarrow} \mathrm{a}\ \underline{B} \underset{4}{\Rightarrow} \mathrm{a}\ \mathrm{b}\ \mathrm{b}$

- nieterminal który w danym kroku jest rozwijany (zastępowany prawą stroną produkcji) jest podkreślony, a numer użytej produkcji – odnotowany pod strzałką

Gramatyki bezkontekstowe: przykład 2

- gramatyka

$$G = \langle \{E, O\}, \{\texttt{num}, \texttt{(}, \texttt{)}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}, P, E \rangle$$

P:

$\underline{E} \to \texttt{num}$	(1)		$O \to \texttt{+}$	(4)
$E \to \texttt{(}\ E\ \texttt{)}$	(2)		$O \to \texttt{-}$	(5)
$E \to E\ O\ E$	(3)		$O \to \texttt{*}$	(6)
			$O \to \texttt{/}$	(7)

- wywód słowa (num - num) z E

$$\underline{E} \underset{2}{\Rightarrow} \texttt{(}\ \underline{E}\ \texttt{)} \underset{3}{\Rightarrow} \texttt{(}\ E\ O\ \underline{E}\ \texttt{)} \underset{1}{\Rightarrow} \texttt{(}\ \underline{E}\ O\ \texttt{num}\ \texttt{)}$$

$$\underset{1}{\Rightarrow} \texttt{(}\ \texttt{num}\ \underline{O}\ \texttt{num}\ \texttt{)} \underset{5}{\Rightarrow} \texttt{(}\ \texttt{num - num}\ \texttt{)}$$

Gramatyki bezkontekstowe

wywód lewo-/prawostronny

- wywód lewostronny to taki, w którym w każdym kroku wybieramy zawsze skrajnie lewy nieterminal:

$$\underline{E} \underset{2}{\Rightarrow} (\ \underline{E}\ ) \underset{3}{\Rightarrow} (\ \underline{E}\ O\ E\ ) \underset{1}{\Rightarrow} (\ \texttt{num}\ \underline{O}\ E\ )$$
$$\underset{5}{\Rightarrow} (\ \texttt{num}\ \texttt{-}\ \underline{E}\ ) \underset{1}{\Rightarrow} (\ \texttt{num}\ \texttt{-}\ \texttt{num}\ )$$

- wywód prawostronny to taki, w którym w każdym kroku wybieramy zawsze skrajnie prawy nieterminal:

$$\underline{E} \underset{2}{\Rightarrow} (\ \underline{E}\ ) \underset{3}{\Rightarrow} (\ E\ O\ \underline{E}\ ) \underset{1}{\Rightarrow} (\ E\ \underline{O}\ \texttt{num}\ )$$
$$\underset{5}{\Rightarrow} (\ \underline{E}\ \texttt{-}\ \texttt{num}\ ) \underset{1}{\Rightarrow} (\ \texttt{num}\ \texttt{-}\ \texttt{num}\ )$$

- formy zdaniowe uzyskiwane w wywodzie lewostronnym z symbolu startowego gramatyki nazywamy lewostronnymi formami zdaniowymi; formy zdaniowe uzyskiwane w wywodzie prawostronnym nazywamy prawostronnymi formami zdaniowymi

Gramatyki bezkontekstowe

język generowany przez gramatykę

- zbiór wszystkich słów zbudowanych z symboli terminalnych, które da się wywieść z symbolu początkowego gramatyki G nazywamy językiem generowanym przez gramatykę G i oznaczamy $L(G)$

$$L(G) = \left\{ \alpha \mid S \stackrel{*}{\Rightarrow} \alpha \right\}$$

Gramatyki bezkontekstowe

równoważność gramatyk

- gramatyki G_1 i G_2 są równoważne, jeśli generują ten sam język

$$L(G_1) = L(G_2)$$

- przykład gramatyk równoważnych

$$G_1: \quad \begin{array}{l} S \to \mathrm{a}\ B \\ S \to B \\ B \to \mathrm{b} \\ B \to \mathrm{bb} \end{array} \qquad G_2: \quad \begin{array}{l} S \to A\ B \\ A \to \mathrm{a} \\ A \to \epsilon \\ B \to \mathrm{b} \\ B \to \mathrm{bb} \end{array}$$

- $L(G_1) = L(G_2) = \{\, \mathrm{ab}\,,\ \mathrm{abb}\,,\ \mathrm{b}\,,\ \mathrm{bb}\,\}$

Gramatyki bezkontekstowe

striktura składniowa

- gramatyka, oprócz tego, że generuje/definiuje zbiór napisów, wiąże z tymi napisami pewną strukturę, zwaną strukturą składniową; struktura ta to drzewo rozbioru

Gramatyki bezkontekstowe

drzewo rozbioru

- drzewo rozbioru (drzewo rozbioru skłaniowego, drzewo wywodu, drzewo wyprowadzenia) opisuje budowę (strukturę składniową) wyrażenia należącego do języka
  - etykietami wierzchołków są symbole nieterminalne, terminalne, bądź ϵ
  - wszystkie wierzchołki nie będące liśćmi etykietowane są symbolami nieterminalnymi
  - etykietą korzenia jest symbol początkowy gramatyki
  - jeśli wierzchołek etykietowany symbolem A ma synów etykietowanych kolejno symbolami $X_1, X_2, \ldots, X_n$, to w gramatyce musi istnieć produkcja $A \rightarrow X_1 X_2 \ldots X_n$
  - wierzchołek etykietowany ϵ musi być liściem i jedynym synem swego ojca

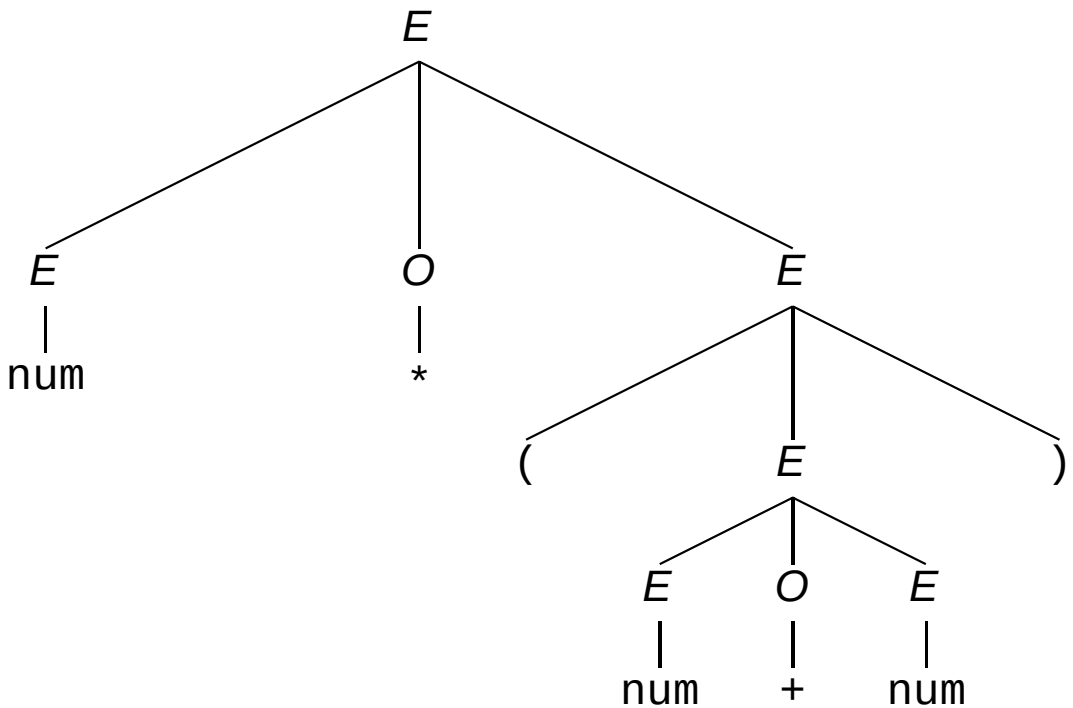
Gramatyki bezkontekstowe

drzewo rozbioru – przykład

- gramatyka G_3

E	$\rightarrow$ num	(1)	O	$\rightarrow$ +	(4)
E	$\rightarrow$ (E)	(2)	O	$\rightarrow$ -	(5)
E	$\rightarrow$ E O E	(3)	O	$\rightarrow$ *	(6)
			O	$\rightarrow$ /	(7)

- drzewo rozbioru dla wyrażenia `num * ( num + num )`



Gramatyki bezkontekstowe

niejednoznaczność

- gramatyka bezkontekstowa jest niejednoznaczna, jeśli pewne słowo ma więcej niż jedno drzewo rozbioru
- ... lub – co równoważne – więcej niż jeden wywód prawostronny (lewostronny)

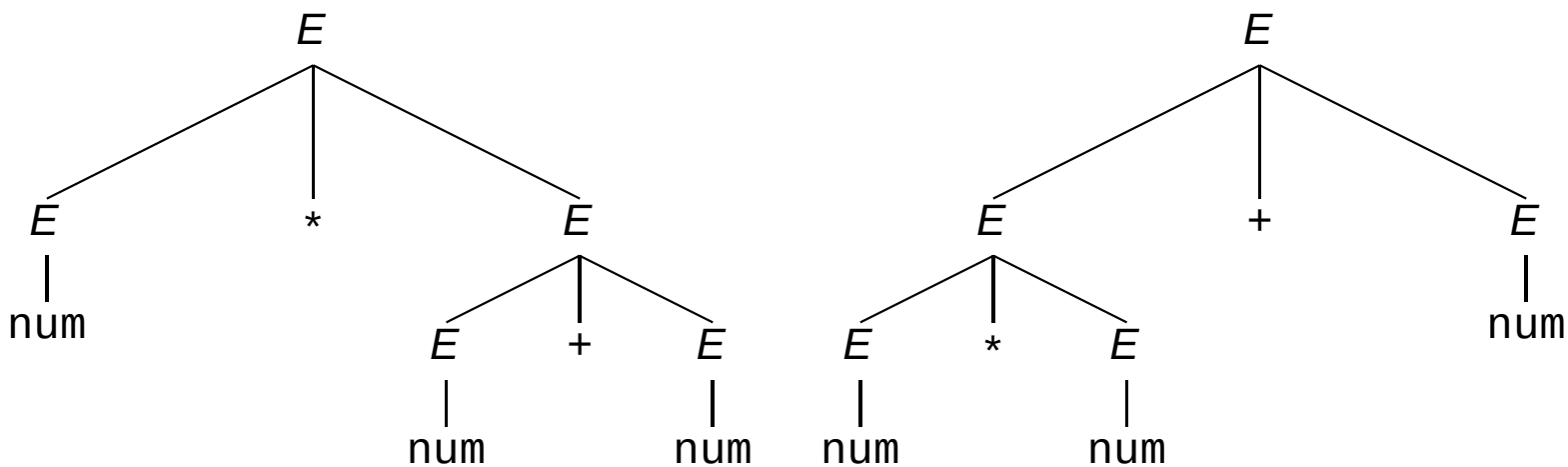
Gramatyki bezkontekstowe

niejednoznaczność – przykład

- gramatyka G_4

$$\underline{E} \to \texttt{num} \qquad \underline{E} \to E * E$$
$$\underline{E} \to E + E \qquad \underline{E} \to E \,/\, E$$
$$\underline{E} \to E - E \qquad \underline{E} \to (\ E\ )$$

- słowo `num * num + num`
- dwa drzewa



Gramatyki bezkontekstowe

usuwanie niejednoznaczności

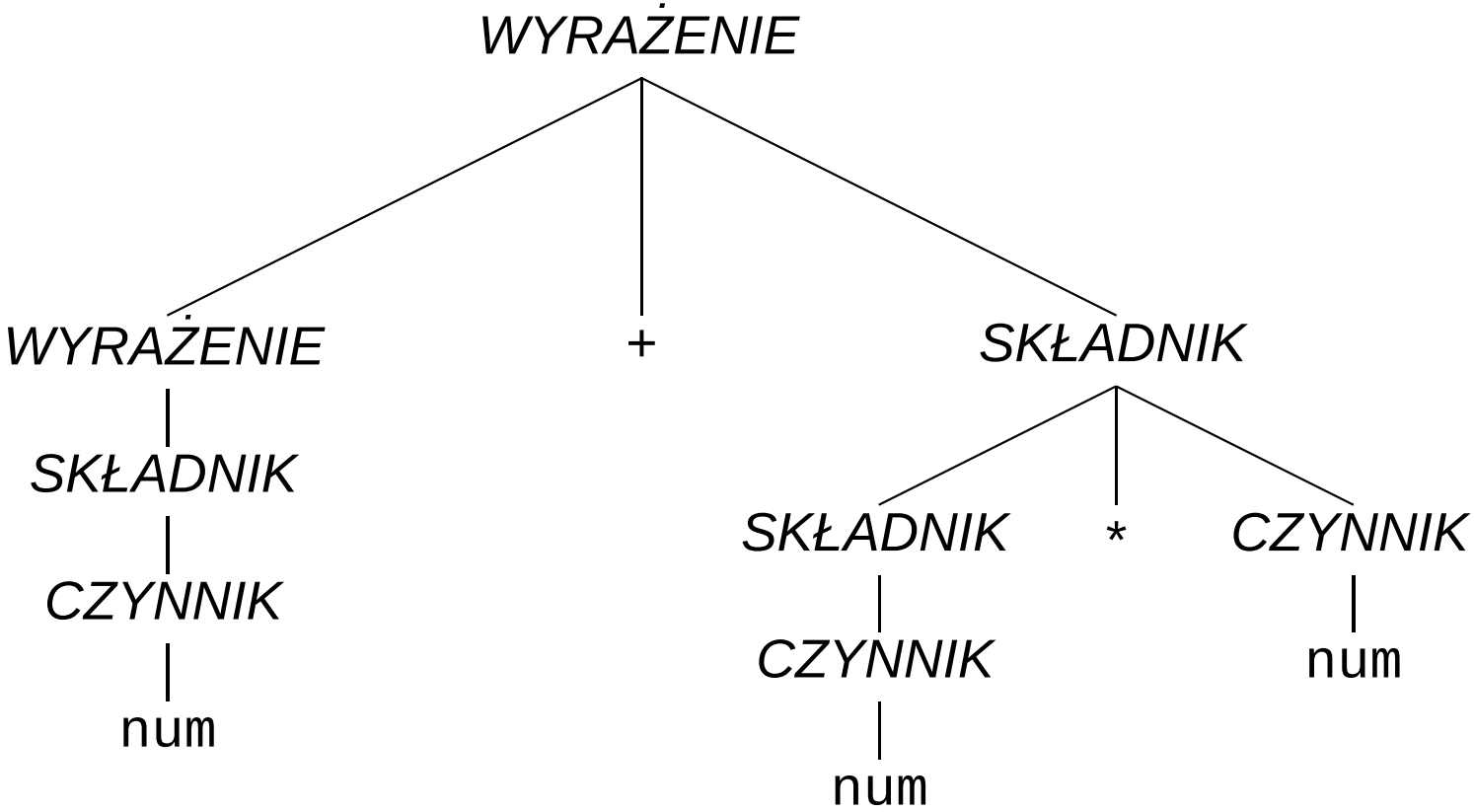
- niejednoznaczność gramatyki często można usunąć formułując ją w inny sposób
- gramatyka jednoznaczna G_5 (równoważna gramatykom G_3 i G_4)

$$
\begin{array}{lcl}
\underline{\textit{WYRAŻENIE}} & \rightarrow & \textit{WYRAŻENIE} + \textit{SKŁADNIK} \\
\underline{\textit{WYRAŻENIE}} & \rightarrow & \textit{WYRAŻENIE} - \textit{SKŁADNIK} \\
\underline{\textit{WYRAŻENIE}} & \rightarrow & \textit{SKŁADNIK} \\
\textit{SKŁADNIK} & \rightarrow & \textit{SKŁADNIK} * \textit{CZYNNIK} \\
\textit{SKŁADNIK} & \rightarrow & \textit{SKŁADNIK} / \textit{CZYNNIK} \\
\textit{SKŁADNIK} & \rightarrow & \textit{CZYNNIK} \\
\textit{CZYNNIK} & \rightarrow & \texttt{num} \\
\textit{CZYNNIK} & \rightarrow & (\ \textit{WYRAŻENIE}\ )
\end{array}
$$

Gramatyki bezkontekstowe

usuwanie niejednoznaczności c.d.

- jedyne drzewo rozbioru dla `num * num + num` w gramatyce G_5



Gramatyki bezkontekstowe

usuwanie niejednoznaczności c.d.

- niejednoznaczność nie zawsze da się usunąć
- istnieją języki bezkontekstowe silnie niejednoznaczne, tzn. takie, które da się opisać wyłącznie niejednoznacznymi gramatykami

Gramatyki bezkontekstowe

quasiterminale

- napisy, które z punktu widzenia składniowego traktujemy tak, jak atomowe symbole terminalne, choć w wyrażeniach języka mogą być różnie reprezentowane, nazywamy quasi-terminalami (termin matematyczny) lub jednostkami leksykalnymi (termin informatyczny, ang. token)
- przykłady: liczba (`12.54`), identyfikator (`main`), stała napisowa (`''Ala ma kota''`).
- w gramatykach, które pojawiały się na wcześniejszych stronach, symbol `num` użyty został w funkcji quasi-terminala

Analiza składniowa

metody podstawowe

podstawowe klasy algorytmów

- algorytmy wstępujące (z dołu do góry, ang. bottom-up) - budują drzewo rozbioru w kierunku od liści do korzenia
- algorytmy zstępujące (z góry na dół, ang. top-down) - budują drzewo rozbioru w kierunku od korzenia do liści
- algorytmy łączące elementy metody wstępującej i zstępującej

Analiza składniowa

algorytm wstępujący – implementacja ze stosem

- wykorzystujemy stos, na który odkładane są symbole terminalne i nieterminalne
- na początku stos jest pusty
- akcje:

  przesunięcie przeniesienie symbolu terminalnego z wejścia na stos

  redukcja zastąpienie na szczycie stosu prawej strony produkcji jej lewą stroną

- sukces, gdy pobierzemy wszystkie symbole z wejścia, a na stosie będzie symbol początkowy gramatyki

Analiza składniowa

algorytm wstępujący ze stosem – konflikty

- konflikt – stan analizatora, w którym możliwe jest wykonanie więcej niż jednej akcji i trzeba dokonać wyboru
- możliwe konflikty:
  - konflikt przesunięcie-redukcja
  - konflikt redukcja-redukcja

Analiza składniowa

algorytm wstępujący ze stosem – przykład przebiegu

- gramatyka (G_4)

$$\underline{E} \rightarrow E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{ - } E \mid \texttt{ (} E \texttt{) } \mid \texttt{num}$$

- wejście `num * num - num`
- możliwy przebieg algorytmu

stos	**wejście**	**akcja**	**konflikt**
$	`num * num - num`	przesunięcie	
$ `num`	`* num - num`	redukcja	p-r
$ *E*	`* num - num`	przesunięcie	
$ *E* `*`	`num - num`	przesunięcie	
$ *E* `* num`	`- num`	redukcja	p-r
$ *E* `* E`	`- num`	redukcja	**p-r**
$ *E*	`- num`	przesunięcie	
$ *E* `-`	`num`	przesunięcie	
$ *E* `- num`		redukcja	
$ *E* `- E`		redukcja	r-r
$ *E*		akceptacja	

Analiza składniowa

algorytm wstępujący ze stosem – przykład przebiegu przesunięcie

- ten sam przebieg algorytmu (tylko inaczej wydrukowany)

stos	wejście	akcja	konflikt
$	num * num - num	przesunięcie	
$num	*** num - num**	redukcja	p-r
$*E*	* num - num	przesunięcie	
$*E* *	num - num	przesunięcie	
$*E* * num	**- num**	redukcja	p-r
$*E* * *E*	- num	przesunięcie	**p-r**
$*E* * *E* -	num	przesunięcie	
$*E* * *E* - num		redukcja	
$*E* * *E* - *E*		redukcja	r-r
$*E* * *E*		redukcja	
$*E*		akceptacja	

- stos+wejście = prawostronna forma zdaniowa
- wyróżnione pogrubieniem prawostronne formy zdaniowe tworzą wywód prawostronny

Analiza składniowa

prefiks żywotny prawostronnej formy zdaniowej

- uchwyt prawostronnej formy zdaniowej to podciąg γ formy zdaniowej $\alpha\gamma\beta$, $\beta \in T^*$, taki że istnieje produkcja $A \rightarrow \gamma$ i $\alpha A\beta$ jest prawostronną formą zdaniową.
- prefiks żywotny prawostronnej formy zdaniowej to prefiks prawostwonnej formy zdaniowej nie wychodzący poza skrajnie prawy uchwyt (uchwytów może być wiele, chodzi o ten najdalej sięgający w prawo)
- podczas każdego przebiegu algorytmu zstępującego ze stosem zakończonego sukcesem, w każdym momencie zawartość stosu musi być żywotnym prefiksem prawostronnej formy zdaniowej.

Analiza składniowa

algorytm LR(1)

- jest deterministyczną wersją algorytmu wstępującego ze stosem
- ”zna” wszystkie możliwe żywotne prefiksy prawostronnych form zdaniowych; tylko one mogą pojawić się na stosie;
- decyzję, jaką akcję wykonać w danym kroku, podejmuje na podstawie stanu stosu i podglądu jednego symbolu wejściowego.
- akcje:

przesunięcie	j.w.
redukcja	j.w.
akceptacja	jeśli na stosie symbol początkowy, a wejście wyczerpane
błąd	jeśli nie można wykonać żadnej innej akcji (w szczególności przesunięcia lub redukcji, w wyniku której na stosie znalazłby się żywotny prefiks)

Analiza składniowa

algorytm LR(1) – przykład przebiegu

- gramatyka G_4

$$\underline{E} \rightarrow E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{ - } E \mid \texttt{ (} E \texttt{) } \mid \texttt{num}$$

- wejście `num * num - num`
- przebieg algorytmu

stos	**wejście**	**akcja**	**konflikt**
$	num * num - num	przesunięcie	
$num	* num - num	redukcja	(nie ma konfliktu p-r)
$E	* num - num	przesunięcie	
$E *	num - num	przesunięcie	
$E * num	- num	redukcja	(nie ma konfliktu p-r)
$E * E	- num	przesunięcie	**p-r**
$E * E -	num	przesunięcie	
$E * E - num		redukcja	
$E * E - E		redukcja	(nie ma konfliktu r-r)
$E * E		redukcja	
$E		akceptacja	

- w krokach opisanych ’(nie ma konfliktu p-r)’ przesunięcie doprowadziłoby do pojawienia się na stosie ciągu symboli, który nie jest żywotnym prefiksem, dlatego w algorytmie LR(1) konflikt nie wystąpi

Analiza składniowa

algorytm LR(1) c.d.

- podstawowy algorytm LR(1) nie może być stosowany dla gramatyki, która prowadzi do powstawania konfliktów, jest to bowiem algorytm deterministyczny
- w praktyce: algorytm LR(1) rozbudowuje się o zasady rozstrzygania konfliktów

yacc

- **yacc** to nazwa językia specyfikacji i zarazem generatora analizatorów składniowych wykorzystujących algorytm LR(1) rozszerzony o mechanizmy rozstrzygania konfliktów
- **bison** jest darmową reimplementacją `yacca` na licencji GNU
- zastosowanie:
  - budowa modułów programowych analizatora składniowego
  - budowa przetworników tekstu sterowanych składnią

yacc

informacje ogólne

- specyfikacja analizatora składniowego tłumaczona jest przez program yacc (bison) na język C i przybiera postać funkcji `yyparse()`
- aby otrzymać program wykonywalny konieczne jest dostarczenie definicji funkcji:
  - `main()`
  - `yyerror(char*)` – funkcja wywoływana w przypadku wystapienia błędu składniowego
  - `yylex()` – analizator leksykalny

yacc

struktura programu

- struktura programu `yacc`owego jest taka sama jak `lex`owego

sekcja deklaracji i definicji

%%

reguły

%%

podprogramy pomocnicze

yacc

struktura programu – sekcja deklaracji i definicji

```
%{
deklaracje i definicje w języku C

%}
deklaracje Yacc
```

yacc

reguły

- reguły są produkcjami gramatyki bezkontekstowej, w których dodatkowo mogą wystąpić akcje
- reguła ma postać:

$$N \ : \ s_1 \ s_2 \ \ldots \ s_n \ ;$$

- N – symbol nieterminalny
- $s_1 \ s_2 \ \ldots \ s_n$ – składniki, z których każdy może być:
  - symbolem terminalnym
  - symbolem nieterminalnym
  - akcją
- reguły o takiej samej lewej stronie można grupować przy użyciu operatora |

Yacc: symbole terminalne

- symbolem terminalnym może być:
  - znak (literał znakowy, np. `'a'`, `'='`)
  - nazwa jednostki leksykalnej (stała symboliczna)
- nazwy jednostek leksykalnych wprowadza się za pomocą deklaracji:
  - `%token` *t*
  - `%left` *t*
  - `%right` *t*
  - `%nonassoc` *t*
- np.

```
%token LICZBA
%nonassoc ASSIGNMENT_OPERATOR
```

yacc

symbole nieterminalne

- symboli nieterminalnych nie deklaruje się
- wszystkie symbole występujące po lewej stronie jakiejś reguły są uznawane za symbole nieterminalne
- symbol początkowy gramatyki wskazuje się (opcjonalnie) za pomocą deklaracji

```
%start s
```

- domyślnie symbolem startowym jest symbol lewej strony pierwszej produkcji

yacc

akcje

- akcja jest ujętym w nawiasy klamrowe kodem w języku C, w którym można używać pewnych predefiniowanych zmiennych i funkcji
- akcje zazwyczaj umieszcza się jako ostatni składnik reguły
- umieszczanie akcji jako pierwszego składnika reguły jest niewskazane
- akcja wykonywana jest, gdy poprzedzający ją ciąg symboli zostanie rozpoznany na wejściu
- akcja kończąca regułę wykonywana jest w momencie redukcji tą regułą
- (Początkujący powinni korzystać tylko z akcji na końcu reguły.)

yacc

sekcja podprogramów pomocniczych

jak w `lexie`

yacc

wejście

- chcąc pobrać symbol wejściowy, funkcja `yyparse()` wywołuje funkcję `yylex()`. Wartością tej funkcji jest liczba całkowita będąca kodem symbolu terminalnego (jednostki leksykalnej).
- wartości zwracane przez funkcję `yylex()` to symbole terminalne w `yaccu`

yacc

przykład 1 (z funkcją yylex( ) pisaną ręcznie)

p1.y

```
%%
ciag   : ciag ’,’ cyfra
       | cyfra
       ;
cyfra  : ’0’
       | ’1’
       ;
%%
#include <stdio.h>

int yylex()
{ int c;
  while((c=getchar())==’ ’ || c==’\t’ || c==’\n’);
  switch(c)
  { case ’0’: return ’0’;
    case ’1’: return ’1’;
    case ’,’: return ’,’;
    case EOF: return 0;
    default : printf("Nieoczekiwany znak.\n"); exit(1); } }

int yyerror(char* s) { printf("%s\n",s); return 1; }

int main() { yyparse(); }
```

yacc

przykład 1prim (z funkcją `yylex()` wygenerowaną przez `lexa`)

p1prim.y

```
%%
ciag    : ciag ’,’ cyfra
        | cyfra
        ;

cyfra   : ’0’
        | ’1’
        ;
%%
#include <stdio.h>

int yyerror(char* s)
{ printf("%s\n",s);
  return 1; }

int main()
{ yyparse(); }
```

p1prim.l

```
%%

0       return ’0’;
1       return ’1’;
\,      return ’,’;
[ \t\n] ;
.       { printf("Nieoczekiwany znak");
          exit(1); }
```

yacc

przykład 1bis (z funkcją yylex( ) wygenerowaną przez lexa)

p1bis.y

```
%token CYFRA

%%
ciag  : ciag ’,’ CYFRA
      | CYFRA
      ;
%%

#include <stdio.h>

int yyerror(char* s)
{ printf("%s\n",s);
  return 1; }

int main()
{ yyparse(); }
```

p1bis.l

```
%{
    #include "p1bis.tab.h"
%}

%%

[01]    return CYFRA;
\,      return ’,’;
[ \t\n] ;
.       { printf("Nieoczekiwany znak");
          exit(1); }
```

p1bis.tab.h (generowany przez yacc -d ...)

```
...
# define CYFRA 257
...
```

yacc

przykład 2 (współpraca z Leksem)

p2.y

```
%token LICZBA
%token ID
%%
wyr   :  wyr '+' wyr
      |  wyr '-' wyr
      |  wyr '*' wyr
      |  wyr '/' wyr
      |  '(' wyr ')'
      |  '-' wyr
      |  LICZBA
      |  ID
      ;
%%
#include <stdio.h>
int yyerror(char* s)
{ printf("%s\n",s);
  return 1; }
int main()
{ yyparse(); }
```

p2.l

```
%{
   #include "p2.tab.h"
%}
%%
[0-9]+  return LICZBA;
[a-z]+  return ID;
"+"     return '+';
"-"     return '-';
"*"     return '*';
"/"     return '/';
"("     return '(';
")"     return ')';
[ \t\n] ;
.       printf("LEXERR\n"); exit(1);
```

p2.tab.h

```
...
# define LICZBA 257
# define ID 258
...
```

yacc

przykład 3 (akcje)

```
%{
    #include <stdio.h>
%}
%token LICZBA
%token ID
%%
wyr  :  wyr '+' wyr printf("1 ");
     |  wyr '-' wyr printf("2 ");
     |  wyr '*' wyr printf("3 ");
     |  wyr '/' wyr printf("4 ");
     |  '(' wyr ')' printf("5 ");
     |  '-' wyr     printf("6 ");
     |  LICZBA      printf("7 ");
     |  ID          printf("8 ");
     ;
%%
...
```

- we: `- LICZBA * ID + LICZBA`
- wy: `7 8 7 1 3 6`

yacc

priorytety

- terminalom i regułom można przypisać priorytet

```
%token LICZBA ID                /* terminale z nieokreślonym priorytetem */
%left '+' '-'                   /* terminale z najniższym priorytetem */
%left '*' '/'
%left NEG                       /* 'sztuczny' terminal z najwyższym priorytetem */

%%

wyr : wyr '+' wyr
    | wyr '-' wyr
    | wyr '*' wyr
    | wyr '/' wyr
    | '-' wyr     %prec NEG     /* priorytet reguły - równy priorytetowi NEG */
    | '(' wyr ')'
    | LICZBA
    | ID
    ;
%%
...
```

yacc

zasady rozstrzygania konfliktów

- Konflikt przesunięcie-redukcja:
  1. wybierz **redukcję** jeśli priorytet reguły wyższy od priorytetu aktualnego symbolu wejściowego; priorytet reguły to priorytet ostatniego symbolu terminalnego w ciele reguły, chyba że przypisano regule priorytet bezpośrednio dyrektywą `%prec`.
  2. wybierz **redukcję** jeśli priorytety są równe, a symbol lewostronnie łączny (zadeklarowany deklaracją `%left`)
  3. wybierz **przesunięcie** w każdym innym przypadku
- Konflikt redukcja-redukcja:
  1. wybierz **redukcję** regułą, która jako pierwsza pojawia się w specyfikacji

yacc

przykład 4

```
%token LICZBA ID
%left '+' '-'
%left '*' '/'
%left NEG
%%
wyr : wyr '+' wyr   {printf("1 ");}
    | wyr '-' wyr   {printf("2 ");}
    | wyr '*' wyr   {printf("3 ");}
    | wyr '/' wyr   {printf("4 ");}
    | '-' wyr       {printf("5 ");} %prec NEG
    | '(' wyr ')'   {printf("6 ");}
    | LICZBA        {printf("7 ");}
    | ID            {printf("8 ");}
    ;
%%
...
```

- we: `- ID * ID + ID + ID`
- wy: `8 6 8 3 8 1 8 1` (bez priorytetów: `8 8 8 8 1 1 3 6`)

yacc

przykład 4 – wyjaśnienie

stos	wejście	konfl	pr.reg.	pr.sym.	łączl.	akcja
$	- ID * ID + ID + ID					przes.
$ -	ID * ID + ID + ID					przes.
$ - ID	* ID + ID +ID					red.
$ - *wyr*	* ID + ID +ID	p-r	3	2	left	red.
$ *wyr*	* ID + ID + ID					przes.
$ *wyr* *	ID + ID + ID					przes.
$ *wyr* * ID	+ ID + ID					red.
$ *wyr* * *wyr*	+ ID + ID	p-r	2	1	left	red.
$ *wyr*	+ ID + ID					przes.
$ *wyr* +	ID + ID					przes.
$ *wyr* + ID	+ ID					red.
$ *wyr* + *wyr*	+ ID	p-r	1	1	left	red.
$ *wyr*	+ ID					przes.
$ *wyr* +	ID					przes.
$ *wyr* + ID						red.
$ *wyr* + *wyr*						red.
$ *wyr*						red.

yacc

wartości semantyczne

- z każdym symbolem związana jest tzw. wartość semantyczna (można jej nie używać)
- w regule: N `:` s_1 s_2 `...` s_n `;`

  `$$` – wartość semantyczna nieterminala N
  `$1, $2, ... $`n – wartości semantyczne składników s_1, s_2, ..., s_n
- wartości semantycznych używa się po to by:
  - móc związać dodatkową informację z symbolami (np. z jednostką leksykalną LICZBA możemy związać informację o jej wartości)
  - móc przekazywać wartości w obrębie struktury składniowej w trakcie analizy
  - móc prowadzić obliczenia w trakcie analizy składniowej

yacc

wartości semantyczne c.d.

- terminalom wartości semantyczne przypisuje analizator leksykalny (przypisując wartość specjalnej zmiennej `yylval`)
- nieterminalom wartości semantyczne przypisuje się w akcjach
- w akcji będącej j-tym składnikiem prawej strony reguły można korzystać z wartości zmiennych \$$i$, $i < j$
- zmiennej \$\$ przypisuje się zwykle wartość w akcji kończącej regułę (domyślnie \$\$=\$1)

yacc

typ wartości semantycznych

- domyślnie typem wszystkich wartości semantycznych jest `int`
- typ wartości semantycznych symboli można zmienić przy pomocy makrodefinicji

```
#define YYSTYPE typ
```

- jeżeli typy wartości semantycznych różnych symboli mają być różne, używamy deklaracji `%union` (patrz. `man/info bison` lub CJN)

yacc

przykład 5 – obliczanie wartości wyrażenia arytmetycznego

kalkulator.y

```
...
%%
wyr0: wyr printf("=%i\n",$1);
    ;

wyr : wyr '+' skl {$$ = $1+$3;}
    | skl         {$$ = $1;}
    ;
skl : skl '*' czy {$$ = $1*$3;}
    | czy         {$$ = $1;}
    ;
czy : '(' wyr ')' {$$ = $2;}
    | LICZBA      {$$ = $1;}
    ;
%%
...
```

kalkulator.l

```
...
%%
[0-9]+  { yylval=atoi(yytext);
          return LICZBA; }
"+"     return '+';
"*"     return '*';
...
```

Algorytm LR

- Algorytm korzysta ze stosu i tablic, definiujących funkcje ACTION i GOTO.
- Na stosie odkładane są na przemian symbole i stany.
- Stan niesie ze sobą informację o przeczytanym dotąd wejściu.
- Funkcja ACTION[*s*,*a*] wyznacza, jaką akcję należy wykonać w stanie *s*, jeśli następnym symbolem na wejściu jest *a*. Możliwe akcje to:
  - przesuń *s*
  - redukuj zgodnie z $A \rightarrow \alpha$
  - akceptuj
  - błąd
- Funkcja GOTO[*s*,*A*] określa stan, do którego należy przejść po redukcji do symbolu *A*.

Algorytm LR

akcje – szczegóły

- konfiguracja poczatkowa: $< s_0 \, , \, a_1 a_2 \ldots a_n \$ >$

Krok:

- konfiguracja przed: $< s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m \, , \, a_i a_{i+1} \ldots a_n \$ >$
- ACTION[s_m,a_i]=przesuń s
  konfiguracja po: $< s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m \, a_i \, s \, , \, a_{i+1} \ldots a_n \$ >$
- ACTION[s_m,a_i]=redukuj zgodnie z $A \rightarrow \alpha$
  konfiguracja po: $< s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} \, A \, s \, , \, a_i a_{i+1} \ldots a_n \$ >$
  - $r =$ długość α,
  - $s = \mathrm{GOTO}[s_{m-r}, A]$
- ACTION[s_m,a_i]=akceptacja
  konfiguracja po: zatrzymanie pracy z wynikiem SUKCES
- ACTION[s_m,a_i]=błąd
  konfiguracja po: zatrzymanie pracy z wynikiem PORAŻKA

Algorytm LR

tworzenie tablic ACTION i GOTO

- metoda SLR
- metoda kanoniczna
- metoda LALR (bison)
- szczegóły – patrz ASU

Algorytm LR

tworzenie tablic metodą SLR

- Rozważamy gramatyke rozszerzoną o produkcję $S' \rightarrow S$
- sytuacja LR(0) – produkcja z kropką w pewnym miejscu prawej strony (np. z produkcji $A \rightarrow B$ c otrzymujemy sytuacje: $[A \rightarrow \cdot B\ \mathrm{c}]$, $[A \rightarrow B \cdot \mathrm{c}]$, $[A \rightarrow B\ \mathrm{c} \cdot]$).
- domknięcie zbioru sytuacji I – najmniejszy zbiór sytuacji, zawierający I, oraz taki, że jeśli $[A \rightarrow \alpha \cdot B\beta]$ należy do I i $B \rightarrow \gamma$ jest produkcją, to do I należy również $[B \rightarrow \cdot\gamma]$.
- przejście(I, X) – domknięcie zbioru wszystkich sytuacji $[A \rightarrow \alpha X \cdot \beta]$ takich, że $[A \rightarrow \alpha \cdot X\beta]$ należy do I.
- kanoniczna rodzina zbiorów sytuacji LR(0) – najmniejsza rodzina C zbiorów sytuacji taka, że: 1) domknięcie $(\{[S' \rightarrow \cdot S]\})$ należy do C, 2) jeżeli zbiór sytuacji I należy do C, to dla każdego symbolu X, przejście (I, X) – o ile nie jest puste – też należy do C.

Algorytm LR

algorytm tworzenia tablic SLR

1. Zbuduj $C = \{I_0, I_1, \ldots, I_n\}$, rodzinę zbiorów sytuacji LR(0) dla G'.
2. stan i odpowiada rodzinie sytuacji I_i:
   - jeśli $[A \to \alpha \cdot a\beta] \in I_i$ i przejście$(I_i, a) = I_j$, a jest terminalem, to ACTION[i,a] przypisz "przesuń j"
   - jeśli $[A \to \alpha \cdot\,] \in I_i$, $A \neq S'$ to ACTION[i,a] przypisz "redukuj według $A \to \alpha$", dla wszystkich $a \in \mathrm{FOLLOW}(A)$ (przez $\mathrm{FOLLOW}(A)$ oznaczamy zbiór terminali, jakie moga pojawić się w wywodach prawostronnych bezposrednio po nieterminalu A. Sposób wyznaczania $\mathrm{FOLLOW}(A)$ – patrz ASU, CJN.)
   - jeśli $[S' \to S\cdot] \in I_i$ to ACTION[i,\$] przypisz "akceptuj"
3. jeśli przejście$(I_i, A) = I_j$, A jest nieterminalem, to GOTO[i,a] przypisz j
4. pozycje tablic, którym nie nadano wartości oznaczamy jako błędne
5. stan startowy odpowiada rodzinie sytuacji zawierającej $[S' \to \cdot\, S]$.

Algorytm LR

algorytm tworzenia tablic SLR

- Jeśli w kroku 2. algorytmu stworzymy sprzeczne akcje, gramatyka nie jest SLR i algorytm nie generuje analizatora.
- Metody kanoniczna i LALR są silniejsze: generują analizatory dla większej klasy gramatyk.
- Konstrukcja tablic LR(1) przebiega bardzo podobnie, zamiast sytuacji LR(0), rozważa się sytuacje LR(1) postaci: [*Produkcja z kropką*, *a*], gdzie *a* jest terminalem, jaki może wystapić zaraz po ciągu opisywanym *Produkcją*.
- szczegóły: patrz ASU, CJN

Algorytm LR

tworzenie tablic SLR (przykład za AHU)

gramatyka:

(0) $E' \to E$
(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \texttt{id}$

rodzina podzbiorów LR(0):

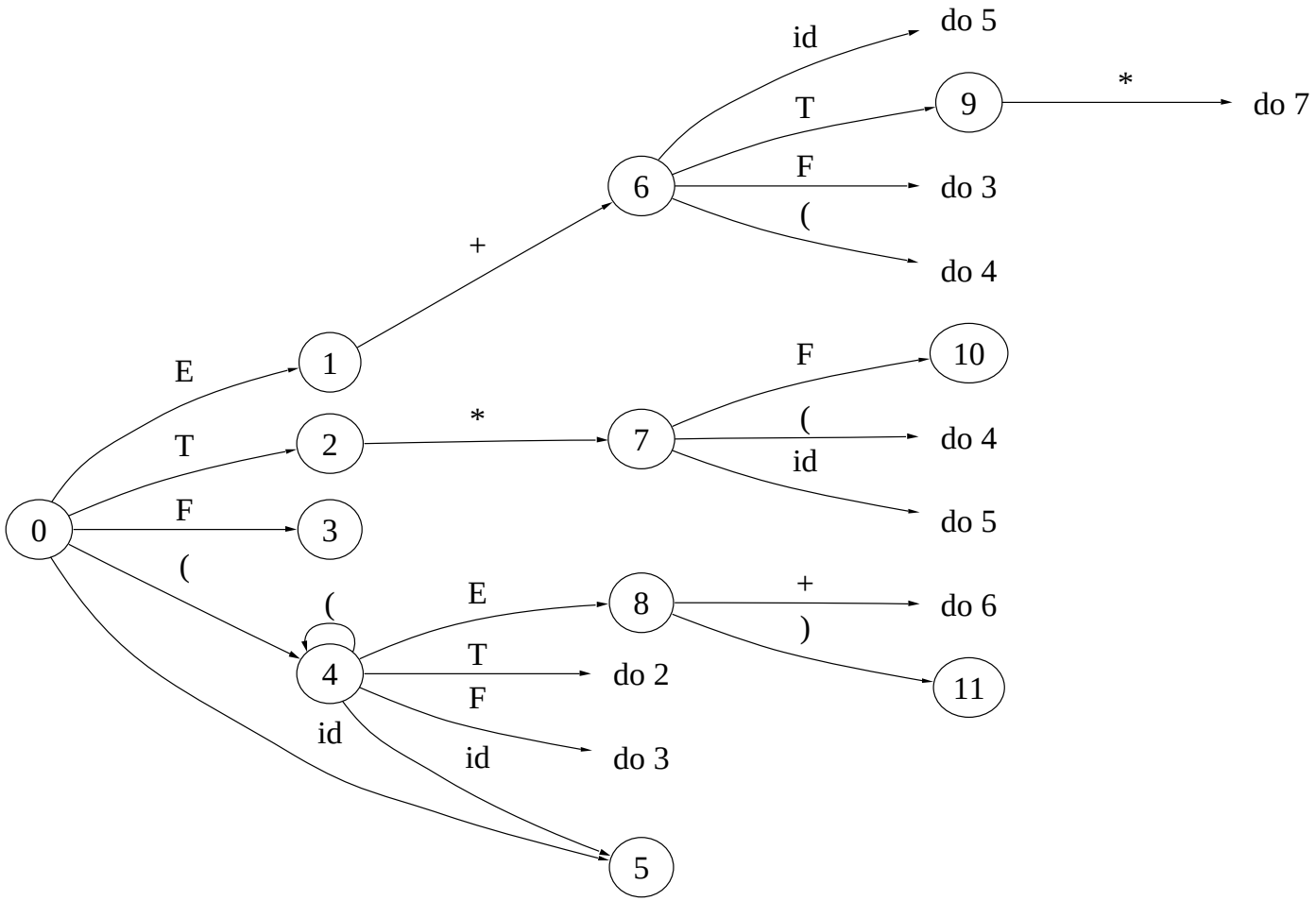
$I_0 = \{ E' \to \cdot E, E \to \cdot E + T, E \to \cdot T, T \to \cdot T * F, T \to \cdot F, F \to \cdot (E), F \to \cdot \texttt{id} \}$
$I_1 = \{ E' \to E \cdot, E \to E \cdot + T, \}$
$I_2 = \{ E \to T \cdot, T \to T \cdot * F, \}$
$I_3 = \{ T \to F \cdot, \}$
$I_4 = \{ T \to (\cdot E), E \to \cdot E + T, E \to \cdot T, T \to \cdot T * F, T \to \cdot F, F \to \cdot (E), F \to \cdot \texttt{id} \}$
$I_5 = \{ F \to \texttt{id} \cdot, \}$
$I_6 = \{ E \to E + \cdot T, T \to \cdot T * F, T \to \cdot F, F \to \cdot (E), F \to \cdot \texttt{id} \}$
$I_7 = \{ T \to T * \cdot F, F \to \cdot (E), F \to \cdot \texttt{id} \}$
$I_8 = \{ F \to (E \cdot), E \to E \cdot + T, \}$
$I_9 = \{ E \to E + T \cdot, T \to T \cdot * F, \}$
$I_{10} = \{ T \to T * F \cdot, \}$
$I_{11} = \{ F \to (E) \cdot, \}$

Algorytm LR

tworzenie tablic SLR – przykład

funkcja przejście:

definiuje automat rozpoznający żywotne prefiksy



Algorytm LR

tablice ACTION i GOTO

	ACTION						GOTO		
stan	`id`	`+`	`*`	`(`	`)`	`$`	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				akc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

s*i* = ”przesuń *i*”

r*i* = ”redukuj *i*-tą produkcją”

Yacc: gramatyki niejednoznaczne

- metody SLR, kanoniczna, i LALR nie generują analizatorów dla gramatyk niejednoznacznych (gramatyki takie nie są LR(1))
- **yacc** tak (priorytety, zasady rozwiązywania konfliktów).

Analiza składniowa

literatura

- J. E. Hopcroft, J. D. Ullman, *Wprowadzenie do teorii automatów, języków i obliczeń*, Wydawnictwo Naukowe PWN, 1994, **(rozdziały 4.–6.)**
- A. V. Aho, R. Sethi, J. D. Ullman, *Kompilatory. Reguły, metody i narzędzia*, Wydawnictwo Naukowo-Techniczne, 2002, **(rozdział 4.)**