Metody Realizacji Języków Programowania

Analiza składniowa

Marcin Benke

MIM UW

11 października 2010

Analiza składniowa

Celem analizy składniowej jest budowa drzewa struktury na podstawie strumienia leksemów.
Dzisiaj:

- Opis składni: gramatyki, BNF
- Wywody, drzewa wywodu
- Wieloznaczność
- Analiza top-down i bottom-up
- Analiza metodą LL(1)

Gramatyki

Formalnie, a gramatyka bezkontekstowa jest czwórką:

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

gdzie

- T jest zbiorem symboli terminalnych (leksemów)
- N jest zbiorem symboli nieterminalnych (rozłącznym z T)
- $S \in N$ jest symbolem początkowym
- P jest zbiorem produkcji postaci $\alpha \to \beta$ gdzie $\alpha \in (T \cup N)^+,\ \beta \in (T \cup N)^*$

Konwencja: symbole nieterminalne oznaczamy zwykle wielkimi literami.

Gramatyka indukuje naturalną relację przepisywania $\to$ na zbiorze $(T \cup N)^*$: napisy mogą być przepisywane zgodnie z produkcjami

BNF

Backus-Naur Form — notacja dla gramatyk bezkontekstowych

- Symbole terminalne są wyróżniane przez użycie cudzysłowów, lub innej czcionki.
- W produkcjach $::=$ zastępuje $\rightarrow$
- Można skrótowo zapisywać zbiory produkcji:

$$E ::= E + T \mid T$$

zamiast

$$E ::= E + T$$
$$E ::= T$$

- Pierwsza produkcja wyznacza symbol startowy.
- Istnieje kilka wariacji BNF.
- Nieterminale są czasem zpaisywane w nawiasach kątowych $<$ i $>$ (np: `<expr>`).

Wywody i drzewa wywodu

Rozważmy gramatykę

$$E \to E + E \mid E - E \mid T$$
$$T \to 0 \mid 1 \ldots \mid 9$$

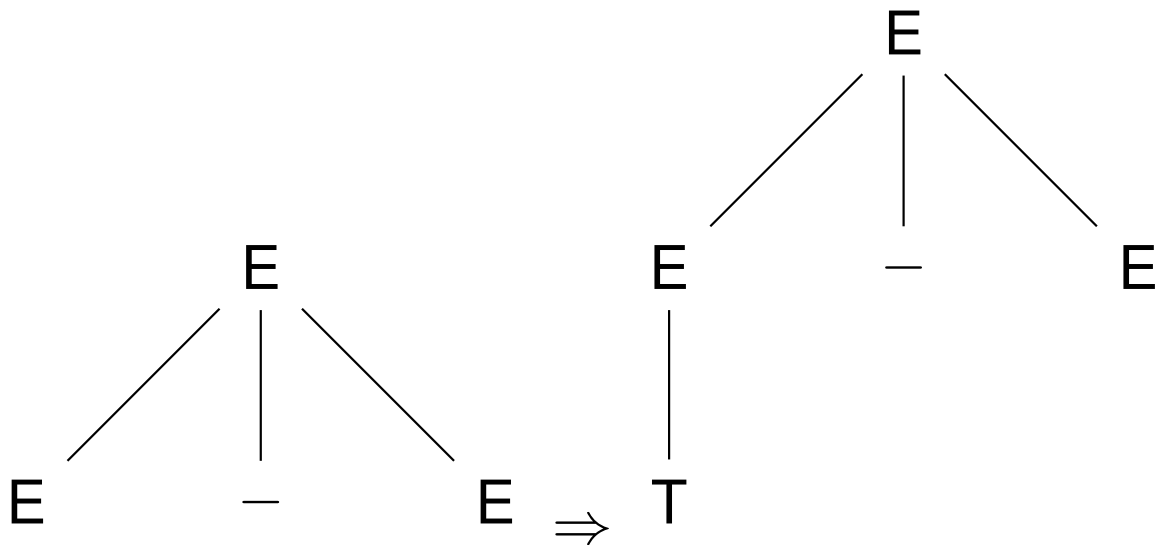
Wyrażenie $1 - 2 + 3$ może być wywiedzione:

$$\begin{aligned} E &\Rightarrow E - E \\ &\Rightarrow T - E \\ &\Rightarrow 1 - E \\ &\Rightarrow 1 - E + E \\ &\Rightarrow 1 - T + E \\ &\Rightarrow 1 - T + T \\ &\Rightarrow 1 - 2 + T \\ &\Rightarrow 1 - 2 + 3 \end{aligned}$$

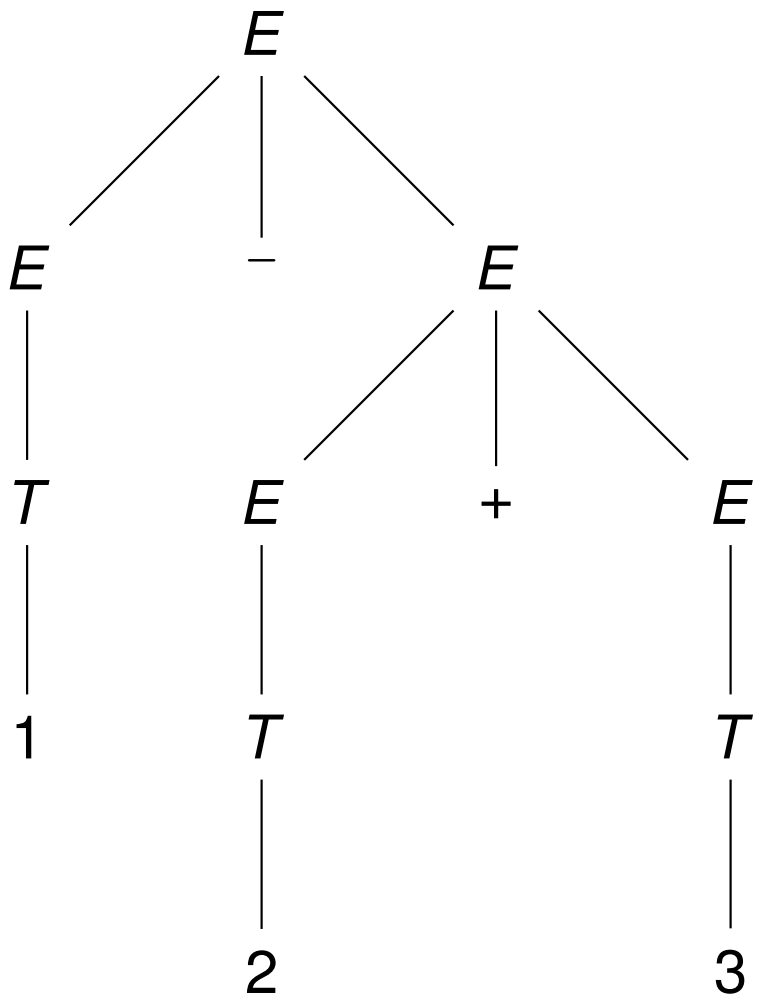
Jest to wywód lewostronny — zawsze przepisujemy najbardziej lewy nieterminal.

Wywody i drzewa wywodu

Wywód możemy przedstawić jako drzewo:



Wywody i drzewa wywodu



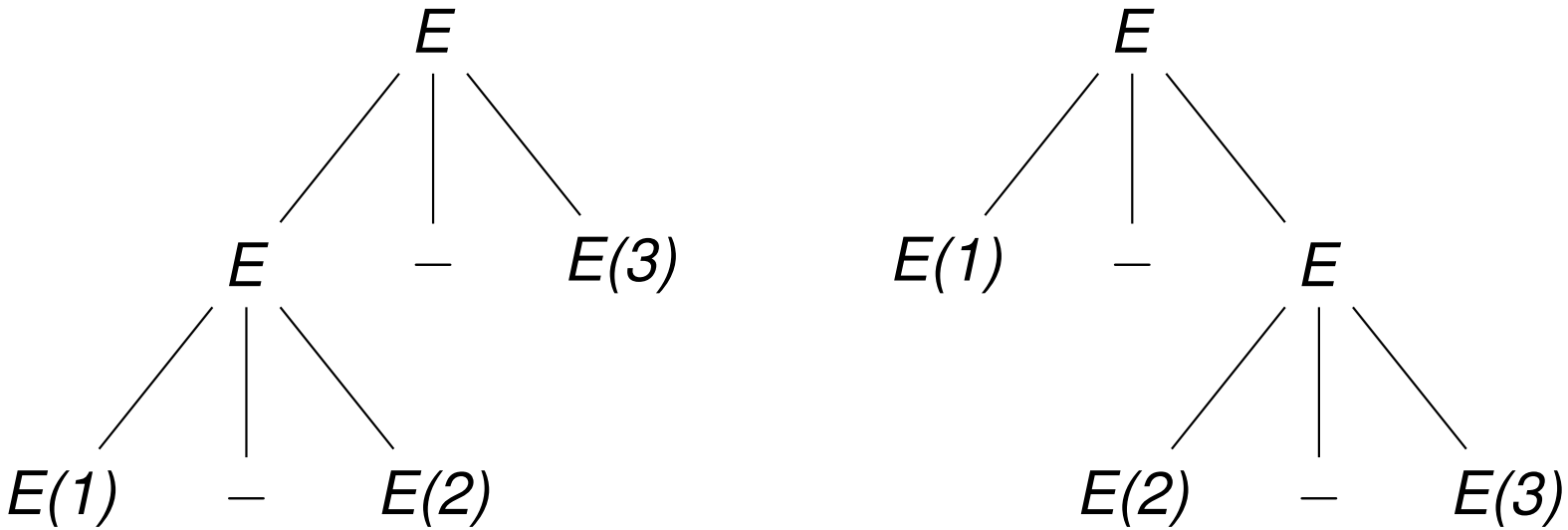
Drzewo wywodu reprezentuje *składnię konkretną*.

Wieloznaczność

Rozważmy gramatykę:

$$E ::= E - E \mid n$$

Istnieją dwa możliwe wywody `1-2-3`:



- Problem: który wywód jest ‘poprawny’?
- ’–’ zwykle łączy w lewo, więc poprawny jest pierwszy wywód.
- Jak możemy uwzględnić ten fakt? Może trzeba nieco zmienić gramatykę?

Łączność

- Możemy “usztywnić” gramatykę, tak aby istniał tylko jeden wywód; w przypadku

$$E ::= E - E \mid T$$

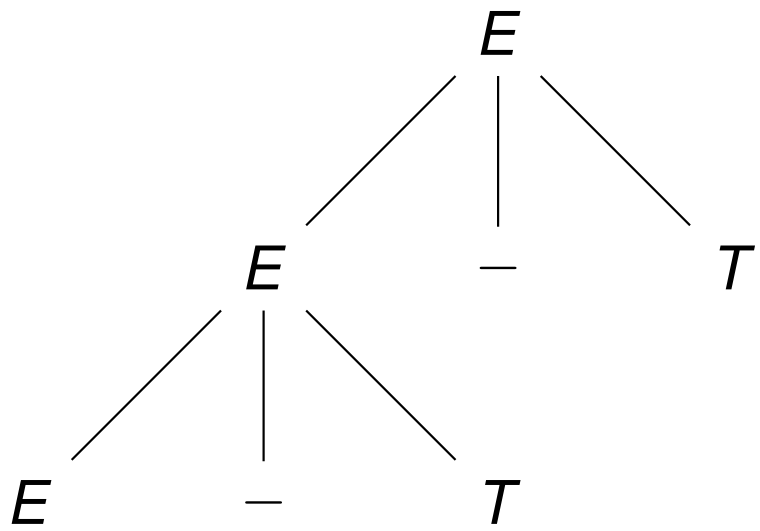
możemy wybrać albo prawe albo lewe E dla drugiego $-$ w naszym przykładzie.

- Chcemy wymusić aby wywód “wyliczył” wszystkie minusy w kolejności występowania

$$E ::= E - T \mid T$$

Otrzymujemy następujące wyprowadzenie:

$$E \Rightarrow E - T \Rightarrow E - T - T \Rightarrow T - T - T \Rightarrow \ldots$$



Dla operatora wiążącego w prawo, np potęgowania, moglibyśmy napisać:

$$E ::= T \hat{} E \mid T$$

Priorytety operatorów

Rozważmy:

$$
\begin{aligned}
E ::= & \; E + T \\
& \mid E * T \\
& \mid T \\
& \ldots
\end{aligned}
$$

Jak możemy opisać (w gramatyce), że ‘*’ wiąże silniej niż ‘+’?

Symbol o niższym priorytecie musi być “bliżej” symbolu startowego:

$$
\begin{aligned}
E &::= E + T \mid T \\
T &::= T * F \mid F \\
F &::= \ldots
\end{aligned}
$$

Wieloznaczność if-then-else

Rozważmy gramatykę:

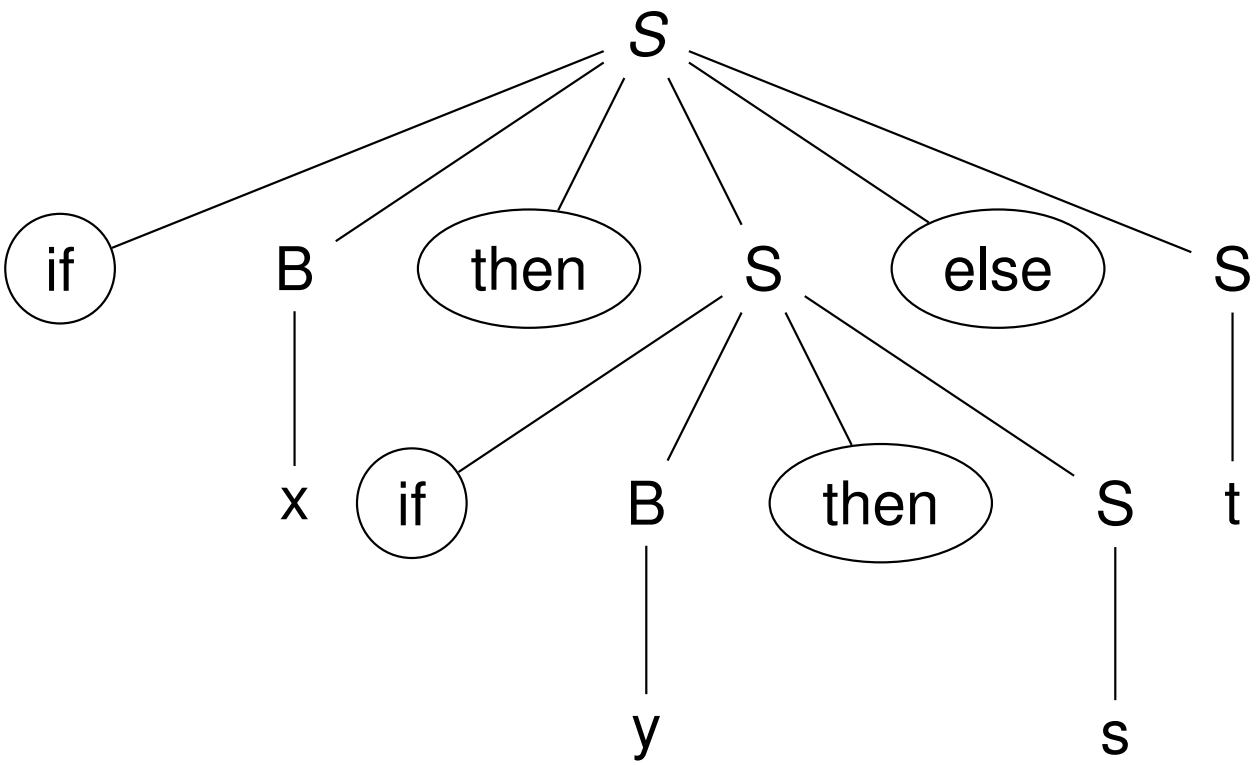
```
S ::= if B then S else S
    | if B then S
    | ...
```

wtedy konstrukcja

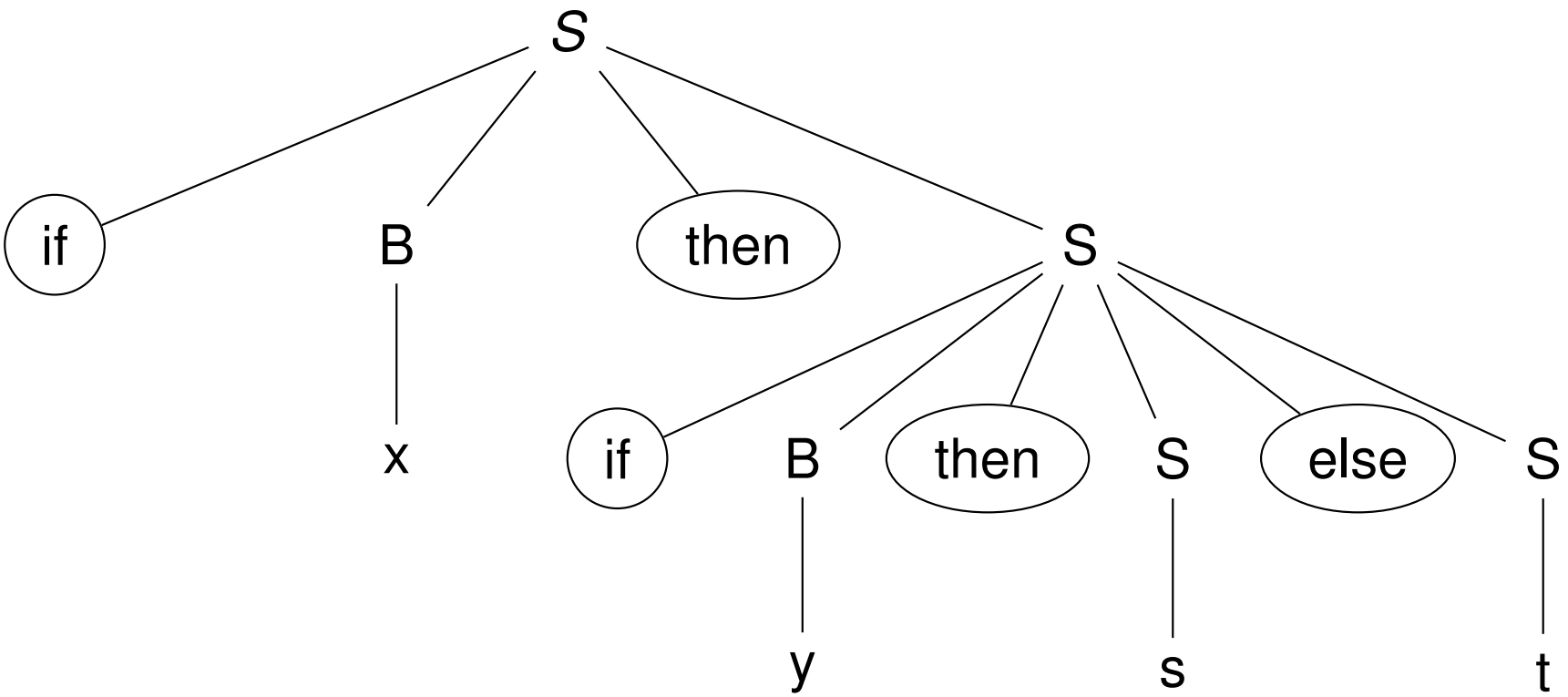
```
if x then if y then s else t
```

Ma dwa możliwe wywody:

Wieloznaczność if-then-else



Wieloznaczność if-then-else



Wieloznaczność if-then-else

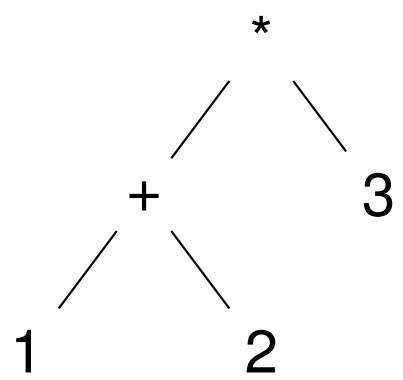
- Możemy rozwiązać problem zmieniając składnię języka:

```
S ::= if B then S else S fi
    | if B then S fi
    | ...
```

- Alternatywnie, możemy rozwiązać problem ad hoc mówiąc, że `else S` zawsze łączy się z najbliższym `if`.
- Można to wyrazić w gramatyce, ale jest to nieco kłopotliwe.

Składnia abstrakcyjna

- Drzewa wywodu są czasem nazywane *drzewami składni konkretnej*
- Zawierają informacje zbędne na dalszych etapach kompilacji
- Zamiast tego potrzebujemy drzew struktury zawierających tylko *informacje semantyczne* (opisujące znaczenie programu).
- Drzewo struktury dla `(1 + 2) * 3`:



Nawiasy należą do składni konkretnej, ale nie do abstrakcyjnej.

Składnia abstrakcyjna

Składnia abstrakcyjna to zbiór typów pozwalających reprezentować strukturę programu. Abstrahujemy od elementów takich jak nawiasy, znaki przestankowe, ...

```
Stmt = SWhile Exp Stmt
     | SAssign Var Stmt
     | SReturn Exp
     | SBlock [Stmt]
```

Analiza syntaktyczna

- Analizator syntaktyczny (*parser*) jest funkcją sprawdzającą, czy dane słowo należy do języka i, jeśli tak, budującą drzewo struktury.
- Algorytm Youngera: $\mathcal{O}(n^3)$ i nie daje drzewa struktury.
- Istnieją efektywne algorytmy dla pewnych klas gramatyk.

Dwa zasadnicze podejścia:

- Top-down: próbujemy sparsować określoną konstrukcję (nieterminal); drzewo struktury budowane od korzenia do liści.
- Bottom-up: W danym napisie znajdujemy możliwe konstrukcje; drzewo budowane od liści do korzenia ze znalezionych kawałków.

Analiza top-down

Schemat analizy top-down możem zapisać jako automat:

- Jeden stan, alfabet stosowy $\Gamma = N \cup T$, akceptacja pustym stosem.
- Na szczycie stosu $a \in T$:
  - jeśli na wejściu a — zdejmij ze stosu, wczytaj następny symbol.
  - wpp — błąd: oczekiwano a.
- Na szczycie stosu $A \in N$, na wejściu a:
  - wybieramy produkcję $A \to \alpha$ taką, że $a \in SELECT(A \to \alpha)$
  - na stosie zastępujemy A przez α

Powyzszy automat ogląda jeden symbol z wejścia, ale łatwo go uogólnić na większą ich liczbę — automat LL(k).

Dla automatu deterministycznego, wybór produkcji jest ważny; zbiór symboli dla których wybieramy produkcję $A \to \alpha$ nazywamy $SELECT(A \to \alpha)$. Teraz zajmiemy się obliczaniem tego zbioru.

Zbiory *FIRST*

Notacja

Niech $w \in T^*$

$$k : w = \begin{cases} a_1 a_2 \dots a_k, & \text{jeśli } w = a_1 a_2 \dots a_k v \\ w\#, & \text{jeśli } |w| < k. \end{cases}$$

(pierwszych k znaków słowa w)

Definicja (*FIRST*)

Niech $w \in (T \cup N)$.

$$FIRST_k(w) = \{\alpha : \exists \beta \in T^*,\ w \to^* \beta,\ \alpha = k : \beta\}$$

(pierwsze k znaków słów wyprowadzalnych z w).

$$FIRST(w) = FIRST_1(w)$$

Zbiory *FOLLOW*

Definicja (*FOLLOW*)

Niech $w \in N$

$$FOLLOW_k(w) = \{\alpha : \exists \beta \in T^*,\ S \rightarrow^* \mu w \beta,\ \alpha = k : \beta\}$$

(pierwsze k znaków mogących wystąpić za w).

Gramatyki LL(k)

Czytając od Lewej, Lewostronny wywód, widzimy (k) symboli.

Definicja

Gramatyka jest LL(k), jeśli dla każdych lewostronnych wyprowadzeń

$$S \rightarrow^* wA\alpha \rightarrow w\beta\alpha \rightarrow^* wx$$
$$S \rightarrow^* wA\alpha \rightarrow w\gamma\alpha \rightarrow^* wy$$

takich, że $FIRST_k(x) = FIRST_k(y)$, *mamy* $\beta = \gamma$

"Jeżeli pierwszych k symboli wyprowadzalnych z A jest wyznaczone jednoznacznie, to także jednoznaczne jest, która produkcja dla A musi być zastosowana w wyprowadzeniu lewostronnym."

Gramatyki silnie LL(k)

Definicja

$$SELECT_k(A \to \alpha) = FIRST_k(\alpha \cdot FOLLOW_k(A))$$

$$SELECT(A \to \alpha) = SELECT_1(A \to \alpha)$$

Definicja

Gramatyka jest silnie LL(k), jeśli dla każdej pary (różnych) produkcji $A \to \alpha$, $A \to \beta$ ich zbiory SELECT są rozłączne.

- Dla gramatyk silnie LL(k) łatwo zbudować parser top-down.
- Każda gramatyka silnie LL(k) jest też LL(k).
- Każda gramatyka LL(1) jest silnie LL(1).

Problemy

Gramatyka nie jest LL(1) jeśli zawiera zbiory produkcji postaci

$$A \to \alpha\beta \mid \alpha\gamma$$

lub produkcje postaci

$$A \to A\beta$$

W drugim przypadku parser się nie zatrzyma!

Gramatykę, w której występują te problemy możemy często przekształcić do równoważnej gramatyki LL(1).

Lewostronna faktoryzacja

Problem pierwszego rodzaju możemy rozwiązać “wyłączając przed nawias” wspólne początki produkcji:

$$A \to \alpha\beta \mid \alpha\gamma$$

zastępujemy przez

$$A \to \alpha Z$$
$$Z \to \beta \mid \gamma$$

gdzie Z jest świeżym nieterminalem.

Eliminacja lewostronnej rekursji

Zbiór produkcji

$$A \to A\alpha \mid \beta$$

zastępujemy

$$A \to \beta R$$
$$R \to \alpha R \mid \varepsilon$$

Na przykład, dla gramatyki

$$E \to E + T \mid T$$

otrzymujemy gramatykę

$$E \to TR$$
$$R \to +TR$$

lub, prościej

$$E \to T + E \mid T$$

Niestety, teraz '+' łączy w prawo, a nie jak chcieliśmy — w lewo.

Wyliczanie FIRST

Dla $t \in T$ mamy $FIRST(t) = \{t\}$.
Dla $A \in N$ mamy:
$A \to \alpha_1 \mid \ldots \mid \alpha_n \Rightarrow FIRST(A) \supseteq FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n)$
Dla $A \to X_1 \ldots X_n$

- $FIRST(A) \supseteq FIRST(X_1) \setminus \{\#\}$
- $X_1 \to^* \varepsilon \Rightarrow FIRST(A) \supseteq FIRST(X_2) \setminus \{\#\}$
- $X_1 X_2 \to^* \varepsilon \Rightarrow FIRST(A) \supseteq FIRST(X_3)$
- $\ldots$
- $X_1 X_2 \ldots X_n \to^* \varepsilon \Rightarrow \# \in FIRST(A)$

Prosty algorytm: zgodnie z powyższymi regułami powiększamy zbiory FIRST tak długo, jak któryś ze zbiorów się powiększa (obliczamy najmniejszy punkt stały).

Wyliczanie *FOLLOW*

Dla każdych $A, X \in N,\ a \in T,\ \alpha, \beta \in (N \cup T)^*$ mamy:

- $A \to \alpha X a \beta \in P$, to $a \in FOLLOW(X)$.
- $A \to \alpha X \in P$, to $FOLLOW(A) \subseteq FOLLOW(X)$
- $A \to \alpha X \beta \in P$, to $FIRST(\beta) \setminus \{\#\} \subseteq FOLLOW(X)$
- $A \to \alpha X \beta \in P$, $\beta \to^* \varepsilon$ to $FOLLOW(A) \subseteq FOLLOW(X)$
- $\# \in FOLLOW(S)$ dla symbolu startowego S.

Prosty algorytm: zgodnie z powyższymi regułami powiększamy zbiory FOLLOW tak długo, jak któryś ze zbiorów się powiększa (obliczamy najmniejszy punkt stały).

Przykład

$E \to E + T$	$\textsc{Select}(E \to E + T) = \textsc{First}(E) = \textsc{First}(T)$
$E \to T$	$\textsc{Select}(E \to T) = \textsc{First}(T) = \textsc{First}(F)$
$T \to T * F$	$\textsc{Select}(T \to T * F) = \textsc{First}(T) = \textsc{First}(F)$
$T \to F$	$\textsc{Select}(T \to F) = \textsc{First}(F) = \{(, \mathbf{a}\}$
$F \to (E)$	
$F \to \mathbf{a}$	

Gramatyka nie jest LL(1).

Transformacja do postaci LL(1)

Usuwamy lewostronną rekursję:

$$E \to TE' \qquad E' \to +TE' \mid \varepsilon$$

$E \to TE'$	niepotrzebne SELECT
$E' \to +TE'$	$\text{SELECT}(E' \to +TE') = \{+\}$
$E' \to \varepsilon$	$\text{SELECT}(E' \to \varepsilon) = \text{FOLLOW}(E') =$ $= \text{FOLLOW}(E) = \{), \#\}$
$T \to FT'$	
$T' \to *FT'$	$\text{SELECT}(T' \to *FT') = \{*\}$
$T' \to \varepsilon$	$\text{SELECT}(T' \to \varepsilon) = \text{FOLLOW}(T') = \mathit{Follow}(T)$ $= \text{FIRST}(E' \cdot \mathit{Follow}(E')) = \{+,), \#\}$
$F \to (E)$	$\text{SELECT}(F \to (E)) = \{(\}$
$F \to \mathbf{a}$	$\text{SELECT}(F \to \mathbf{a}) = \{\mathbf{a}\}$

Dla miłośników algorytmiki

Efektywny algorytm wyliczania FIRST/FOLLOW:

1. Budujemy graf G odpowiedniej relacji na N: dla FIRST "bycia na początku" produkcji, dla FOLLOW — "bycia na końcu".
2. Obliczamy acykliczną kondensację G (graf silnie spójnych składowych).
3. Sortujemy topologicznie.
4. Obliczamy przybliżenia odpowiednich zbiorów (tak jak w jednym obrocie pętli "naiwnego" algorytmu).
5. Dla każdej silnie spójnej składowej przybliżenie rozszerzamy do sumy przybliżeń dla wszystkich jej wierzchołków.
6. Wynikowy zbiór jest sumą przybliżeń dla wszystkich poprzedzających składowych i bieżącej składowej.