Języki i Paradygmaty Programowania

Analiza leksykalna
Analiza składniowa wstępująca (bottom-up)

Marcin Benke

MIM UW

11 kwietnia 2016

Analiza leksykalna

- Prowadzenie analizy składniowej na poziomie pojedynczych znaków jest żmudne i nieefektywne.
- Gramatyki języków naturalnych opisuje się na poziomie słów a nie pojedynczych liter.
- Podobnie w wypadku języków programowania wyróżniamy "słowa" zwane leksemami.
- Definicja gramatyki i analiza składniowa odbywa się na poziomie leksemów.
- Analizator leksykalny (skaner) przetwarza ciąg znaków na ciąg leksemów.

Podstawowe rodzaje leksemów

- Słowa kluczowe, np. `for`, `do`
- Identyfikatory, np. `Foo`, `foo`
- Operatory, np. `++`, `>>=`
- Literały, np `3.14`, `'\n'`, `u"Gżegżółka"`
- "znaki przestankowe", np. `{`, `;`

Przykład

```
int sumto(int n)
{
  int i, sum;
  i = 0;
  sum = 0;
  while (i<n) {
    i = i+1;
    sum = sum+i;
 }
 return sum;
}
```

Analiza leksykalna

Dzielimy tekst na *leksemy*:

int	**id** sumto	(	**int**	**id** n	)	{	
int	**id** i	,	**id** sum	;			
id i	=	**num** 0	;				
id sum	=	**num** 0	;				
while	(	**id** i	<	**id** n	)	{	
id i	=	**id** i	+	**num** 1	;		
id sum	=	**id** sum	+	**id** i	;		
}	**return**	**id** sum	;	}			

Leksemy

- Niektóre leksemy mają wartość semantyczną, np. $\boxed{\textbf{id}\ \mathrm{i}}$, $\boxed{\textbf{num}\ 1}$
- Można opisać przy pomocy wyrażeń regularnych, np.

```
identyfikator = [a-zA-Z][a-zA-Z0-9]*
```

- Można rozpoznać przy pomocy DFA.

Reguła najdłuższego dopasowania

Zwykle wyrażenia regularne opisujące leksemy pozwalają na kilka różnych podziałów tekstu na leksemy, np.

- `--a` $\mapsto$ $\boxed{-}$ $\boxed{-}$ $\boxed{\textbf{id}\ \mathsf{a}}$ lub $\boxed{\text{--}}$ $\boxed{\textbf{id}\ \mathsf{a}}$
- `j23` $\mapsto$ $\boxed{\textbf{id}\ \text{j}}$ $\boxed{\textbf{num}\ 23}$ lub $\boxed{\textbf{id}\ \text{j23}}$

Zwykle wybieramy *najdłuższe możliwe dopasowanie*.
Drugi przykład pokazuje wyraźnie dlaczego.

Odstępy

Znaki takie jak spacja, TAB, koniec linii zwykle są pomijane, mogą jednak służyć do rozdzielania leksemów, np.

`a--3` vs `a - -3`

Leksem może przechowywać swoją pozycję (wiersz, kolumna) w pliku źródłowym, np. dla celów raportowania błędów.

W niektórych językach pozycje niektórych leksemów mogą wpływać na rozbiór i znaczenie programu (np. Python, Haskell)

Generatory analizatorów leksykalnych

- Pisanie analizatora leksykalnego jest zwykle żmudne.
- Dlatego przeważnie jest on generowany automatycznie przez narzędzia takie jak Flex (C,C++), JLex (Java), Alex (Haskell), Ocamllex, C#Lex,...
- Generują one program realizujący automat rozpoznający leksemy na podstawie opisu złożonego z wyrażeń regularnych i przypisanych im akcji.

Analiza wstępująca — metoda LR

- Od Lewej, pRawostronne wyprowadzenie (w odwrotnej kolejności)
- Automat ze stosem, na stosie ciąg terminali i nieterminali
- Jeśli na stosie jest prawa strona produkcji, możemy ją zastąpić symbolem z lewej (redukcja)
- Pytanie, kiedy to robić — poznamy różne techniki.
- Automat startuje z pustym stosem i akceptuje, gdy całe wejście zredukuje do symbolu startowego.

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F$, $4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)

Przykład

Gramatyka: $1 : E \rightarrow E + T$ $2 : E \rightarrow T, 3 : T \rightarrow T * F$, $4 : T \rightarrow F, 5 : F \rightarrow n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \rightarrow n$)

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F,\ 4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$

Przykład

Gramatyka: $1 : E \to E + T$ $2 : E \to T, 3 : T \to T * F,\ 4 : T \to F, 5 : F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$

Przykład

Gramatyka: $1 : E \to E + T$ $2 : E \to T, 3 : T \to T * F,\ 4 : T \to F, 5 : F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F,$ $4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F,\ 4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F,\ 4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$
E + F	* 3	reduce 4:$T \to F$

Przykład

Gramatyka: $1 : E \to E + T$ $2 : E \to T, 3 : T \to T * F,\ 4 : T \to F, 5 : F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$
E + F	* 3	reduce 4:$T \to F$
E + T	* 3	shift (dlaczego?)

Przykład

Gramatyka: $1 : E \to E + T$ $2 : E \to T, 3 : T \to T * F,\ 4 : T \to F, 5 : F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$
E + F	* 3	reduce 4:$T \to F$
E + T	* 3	shift (dlaczego?)
E + T *	3	shift

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F,\ 4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$
E + F	* 3	reduce 4:$T \to F$
E + T	* 3	shift (dlaczego?)
E + T *	3	shift
E + T * 3	#	reduce 5:$F \to n$

Przykład

Gramatyka: $1: E \to E + T$ $2: E \to T, 3: T \to T * F,\ 4: T \to F, 5: F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$
E + F	* 3	reduce 4:$T \to F$
E + T	* 3	shift (dlaczego?)
E + T *	3	shift
E + T * 3	#	reduce 5:$F \to n$
E + T * F	#	reduce 3:$T \to T * F$

Przykład

Gramatyka: $1: E \rightarrow E + T$ $2: E \rightarrow T, 3: T \rightarrow T * F,\ 4: T \rightarrow F, 5: F \rightarrow n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \rightarrow n$)
F	+ 2 * 3	reduce 4:$T \rightarrow F$
T	+ 2 * 3	reduce 2:$E \rightarrow T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \rightarrow n$
E + F	* 3	reduce 4:$T \rightarrow F$
E + T	* 3	shift (dlaczego?)
E + T *	3	shift
E + T * 3	#	reduce 5:$F \rightarrow n$
E + T * F	#	reduce 3:$T \rightarrow T * F$
E + T	#	reduce 1

Przykład

Gramatyka: $1 : E \to E + T$ $2 : E \to T, 3 : T \to T * F,\ 4 : T \to F, 5 : F \to n$

Stos	Wejście	Akcja
(pusty)	1 + 2 * 3	shift (przesuń z wejścia na stos)
1	+ 2 * 3	reduce 5 (redukcja produkcji 5:$F \to n$)
F	+ 2 * 3	reduce 4:$T \to F$
T	+ 2 * 3	reduce 2:$E \to T$
E	+ 2 * 3	shift
E +	2 * 3	shift
E + 2	* 3	reduce 5:$F \to n$
E + F	* 3	reduce 4:$T \to F$
E + T	* 3	shift (dlaczego?)
E + T *	3	shift
E + T * 3	#	reduce 5:$F \to n$
E + T * F	#	reduce 3:$T \to T * F$
E + T	#	reduce 1
E	#	accept

Gramatyki LR(k)

Nieformalnie: jeśli dla formy zdaniowej αw mamy już na stosie α, to para $\langle \alpha, k : w \rangle$ wyznacza jednoznacznie co zrobić, a w szczególności:

- czy na szczycie stosu jest prawa strona jakiejś produkcji? (łatwe, ale może być więcej niż jedna)
- czy należy redukować, a jeśli tak, to którą produkcję? (trudne, podglądamy k symboli z wejścia)

W praktyce ograniczamy się do $k \leqslant 1$.

Kiedy redukować?

Różne metody:

LR(0) — redukujemy kiedy się tylko da
w praktyce za słaba

LR(1) — precyzyjnie wyliczamy dla jakich terminali na wejściu redukować
bardzo silna metoda, ale koszt generowania rzędu 2^{n^2}.

SLR(1) — Simple LR(1): LR(0) + prosty pomysł: redukujemy $A \to \alpha$ jeśli terminal z wejścia należy do $\textsc{Follow}(A)$.

LALR(1) — Look Ahead LR(1): zgrubnie wyliczamy (budujemy automat LR(1) i sklejamy podobne stany).
w praktyce dostatecznie silna metoda,
tyle samo stanów co w automacie LR(0)

Problemy

1. Czy na szczycie stosu jest prawa strona jakiejś produkcji? (łatwe, ale może być więcej niż jedna)
2. Czy należy redukować, a jeśli tak, to którą produkcję?

Tworzymy deterministyczny automat ze stosem, symulujący prawostronne wyprowadzenie.
Automat wykrywa uchwyty (produkcje wraz z miejscem wystąpienia).

Jak rozpoznać uchwyt?

Zbudujemy automat skończony rozpoznający wiele wzorców (możliwe prawe strony produkcji)

Sytuacja LR(0)

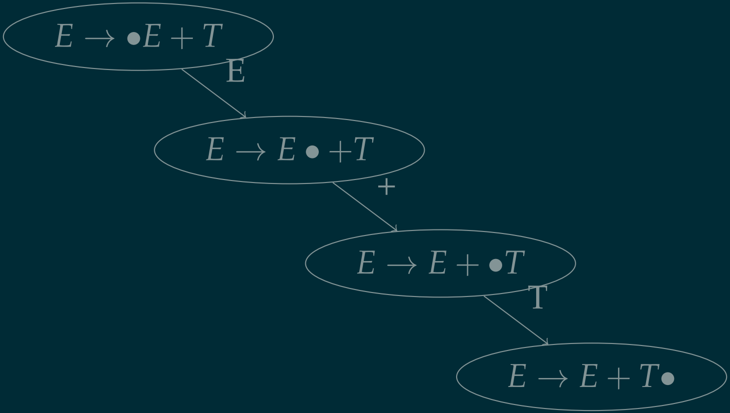
$$A \to \alpha \bullet \beta$$

czyli produkcja z wyróżnionym miejscem.
Jesteśmy w trakcie rozpoznawania $A \to \alpha\beta$,
na stosie jest już α, trzeba jeszcze rozpoznać β

Sytuacja $A \to \beta\bullet$ oznacza, że na stosie jest cała prawa strona produkcji.

Przykład (fragment automatu)

Jeśli mamy sytuację $A \to \alpha \bullet X\beta$, to po rozpoznaniu X mamy sytuację $A \to \alpha X \bullet \beta$



Stany i przejścia automatu LR

- Ponieważ trzeba równocześnie rozpoznawać wiele uchwytów, to stanami automatu będą **zbiory sytuacji**.
- Jeśli jesteśmy w sytuacji $B \to \alpha \bullet A\beta$, to jesteśmy też w sytuacji $A \to \bullet\gamma$ dla każdego $A \to \gamma \in P$
- Zbiór sytuacji musi być domknięty zwn tę implikację:
- $Closure(Q)$ – najmniejszy zbiór zawierający Q oraz taki, że jeśli $B \to \alpha \bullet A\beta \in Closure(Q)$,to

$$\forall A \to \gamma \in P \quad A \to \bullet\gamma \in Closure(Q)$$

- Jeśli $A \to \alpha \bullet X\gamma \in Q$ dla pewnego $X \in N \cup T$, to ze stanu Q jest przejście (po X) do stanu $Closure(A \to \alpha X \bullet \gamma)$

Przykład

$S \to E, E \to E + T \mid E - T \mid T, T \to T * F \mid F, F \to n$

Domknięciem stanu $S \to \ \bullet E$ jest stan zawierający również

Przykład

$S \to E, E \to E + T \mid E - T \mid T, T \to T * F \mid F, F \to n$

Domknięciem stanu $S \to \bullet E$ jest stan zawierający również

$$E \to \bullet E + T, E \to \bullet E - T, E \to \bullet T$$

Przykład

$S \to E, E \to E + T \mid E - T \mid T, T \to T * F \mid F, F \to n$

Domknięciem stanu $S \to \bullet E$ jest stan zawierający również

$$E \to \bullet E + T, E \to \bullet E - T, E \to \bullet T$$

$$T \to \bullet T * F, T \to \bullet F$$

Przykład

$S \to E, E \to E + T \mid E - T \mid T, T \to T * F \mid F, F \to n$

Domknięciem stanu $S \to \bullet E$ jest stan zawierający również

$$E \to \bullet E + T, E \to \bullet E - T, E \to \bullet T$$

$$T \to \bullet T * F, T \to \bullet F$$

$$F \to \bullet n$$

Przykład

$S \to E, E \to E + T \mid E - T \mid T, T \to T * F \mid F, F \to n$

Domknięciem stanu $S \to \bullet E$ jest stan zawierający również

$$E \to \bullet E + T, E \to \bullet E - T, E \to \bullet T$$

$$T \to \bullet T * F, T \to \bullet F$$

$$F \to \bullet n$$

Z tego stanu po rozpoznaniu E przejdziemy do stanu zawierającego domknięcie zbioru

$$S \to E \bullet, E \to E \bullet + T, E \to E \bullet - T$$

(ale już bez $E \to \bullet T$).

Działanie automatu LR

- Dwie tablice indeksowane stanami i symbolami: ACTION (dla terminali) i GOTO (dla nieterminali)
- Stos zawiera stany przetykane symbolami gramatyki
- Automat startuje ze stosem zawierającym symbol początkowy
- Niech na stosie stan s, na wejściu terminal a:
  - $\mathrm{ACTION}[s, a] = \mathbf{shift}\ p$
    przenosi a z wejścia na stos i nakrywa stanem p
  - $\mathrm{ACTION}[s, a] = \mathbf{reduce}(A \to \alpha)$
    zdejmuje $|\alpha|$ par ze stosu
    odsłoni się stan q (zawierał sytuację $\ldots \bullet A \ldots$)
    wkłada na stos A, $\mathrm{GOTO}[q, A]$.
  - Specjalne akcje: **error**, **accept**

Konstrukcja automatu LR

1. Rozszerzamy gramatykę o produkcję $Z \to S\#$ (nowy symbol początkowy)
2. Budujemy automat skończony:
   - stanami są zbiory sytuacji
   - stan początkowy: $\mathit{Closure}(\{Z \to \ \bullet S\#\})$
   - dla stanu p przejście po symbolu X do stanu

   $$\delta(p, X) = \mathit{Closure}(\{A \to \alpha X \bullet \gamma : A \to \alpha \bullet X\gamma \in p\})$$

   - stanem akceptującym jest $\{Z \to S\# \bullet\}$
3. Wypełniamy tablicę sterującą automatu ze stosem.

$$L \to L; s \mid \epsilon \qquad S \to a \mid \epsilon$$

Wypełnianie tablic sterujących

Numerujemy stany, numerujemy produkcje.
Jednolicie dla wszystkich klas automatów wpisujemy akcje **shift** (przepisujemy przejścia automatu skończonego) i **accept**:

Dla przejścia $p \xrightarrow{X} q$ wpisujemy:

- jeśli X jest *terminalem* to

$$\mathrm{ACTION}[p, x] = \textbf{shift q}$$

- jeśli X jest *nieterminalem* to

$$\mathrm{GOTO}[p, x] = \textbf{q}$$

- Jeśli stan p zawiera $S' \rightarrow S \bullet \#$, to $\mathrm{ACTION}[p, \#] = \textbf{accept}$

Redukcje

Tu postępujemy różnie dla różnych klas automatów.

Jeśli stan p zawiera $A \to \alpha \bullet$, to:

LR(0) wpisujemy **reduce**$(A \to \alpha)$ do ACTION$[p, a]$ dla wszystkich a

SLR(1) wpisujemy **reduce**$(A \to \alpha)$ do ACTION$[p, a]$ dla $a \in \textsc{Follow}(A)$

Miejsca nie wypełnione oznaczają **error**

Jeśli gdzieś zostanie wpisana więcej niż jedna akcja, to źle: gramatyka nie jest odpowiedniej klasy (konflikt shift-reduce lub reduce-reduce).

Generatory parserów

- Ręczne tworzenie automatu LR jest w praktyce zbyt żmudne
- Może to z powodzeniem wykonać specjalny program
- Wejście: gramatyka translacyjna (gramatyka + akcje)
- Wyjście: analizator składniowy LR w języku docelowym
- Istnieją dla wielu języków: C,C++ (Yacc,Bison), Java (Cup), Haskell (Happy), Ocaml (Ocamlyacc), …

Happy

```
%name calc
%tokentype { Token }
%token
      int                { TokenInt $$ }
      '+'                { TokenPlus }
      '*'                { TokenTimes }
%%
Exp
      : int                   { EInt $1 }
      | Exp Exp '+'           { EAdd $1 $2 }
      | Exp Exp '*'           { EMul $1 $2 }
```

BNF Converter

- Kompleksowy generator parserów
- Wejście: etykietowana gramatyka BNF
- Wyjście: skaner, parser, pretty-printer, typy dla drzewa struktury, szkielet programu, dokumentacja języka, Makefile,...
- Działa dla wielu języków:
  - Haskell
  - Java
  - C
  - C++
  - O'Caml
- BNFC generuje definicje dla odpowiedniego generatora parserów (i lekserów) na podstawie wejściowej gramatyki LBNF.

Tu w skrócie, więcej — `http://bnfc.digitalgrammars.com/`

LBNF — Labelled BNF

Notacja BNF wzbogacona o informacje o sposobie tworzenia drzewa struktury:

```
EPlus.  Exp  ::= Exp  "+" Exp ;
EInt.   Exp ::= Integer ;
```

```
ben@marcin$ bnfc -m ../exp.cf
The BNF Converter, 2.4b...
writing file Absexp.hs
writing file Lexexp.x (Use Alex 2.0 to compile.)
writing file Parexp.y (Tested with Happy 1.15)
writing file Docexp.tex
writing file Docexp.txt
writing file Skelexp.hs
writing file Printexp.hs
writing file Testexp.hs
writing file Makefile
```

Wygenerowana składnia abstrakcyjna

```
module Absexp where
data Exp = EPlus Exp Exp
         | EInt Integer
         deriving (Eq,Ord,Show)

module Skelexp where
failure :: Show a => a -> Result
transExp :: Exp -> Result
transExp x = case x of
  EPlus exp0 exp  -> failure x
  EInt n  -> failure x
```

LBNF — priorytety

Standardowa transformacja gramatyki uzględniająca priorytety i łączność

```
EPlus.  Exp    ::= Exp "+" Term ;
ETerm.  Exp    ::= Term ;
TTimes. Term   ::= Term "*" Factor ;
TFact.  Term   ::= Factor ;
FInt.   Factor::= Integer ;
```

Da zbyt rozgadaną składnię abstrakcyjną:

```
data Exp = EPlus Exp Term
         | ETerm Term deriving (Eq,Ord,Show)

data Term = TTimes Term Factor
          | TFact Factor deriving (Eq,Ord,Show)

data Factor = FInt Integer deriving (Eq,Ord,Show)
```

LBNF — koercje

Koercje pozwalają wskazać które reguły należą do składni konkretnej:

```
EPlus.  Exp  ::= Exp  "+" Exp2 ;
ETimes. Exp2 ::= Exp2 "*" Exp3 ;
EInt.   Exp3 ::= Integer ;
_.      Exp  ::= Exp2 ;
_.      Exp2 ::= Exp3 ;
_.      Exp3 ::= "(" Exp ")" ;
```

Teraz skladnia abstrakcyjna jest taka jak oczekiwana:

```
data Exp =
    EPlus Exp Exp
  | ETimes Exp Exp
  | EInt Integer
```

LBNF — definicje leksykalne

Leksemy można definiować przy pomocy wyrażeń regularnych:

```
token UIdent (upper (letter | digit | '_')*) ;
```

Predefiniowane leksemy:

```
Integer digit+
Double digit+ '.' digit+ ('e' '-'? digit+)?
 Char '\''((char - ["'\\"]) | ('\\' ["'\\nt"]))'\''
String '"'((char - ["\"\\"]) | ('\\' ["\"\\nt"]))*'"'
 Ident letter (letter | digit | '_' | '\'')*
```

Mają wartość semantyczną odpowiedniego typu (Ident — napis).

LBNF — sekwencje

Wyrażenie w czystmy BNF sekwencji (instrukcji, funkcji,...) jest możliwe, ale żmudne. W LBNF możemy zapisać, że program jest listą funkcji, a funkcje...

```
Prog. Program ::= [Function] ;
Fun. Function ::= Type Ident "(" [Decl] ")"
                  "{" [Stm] "}" ;
```

Sekwencje mają separatory lub terminatory:

```
terminator Function "" ;
terminator Stm "" ;
separator Decl "," ;
```

Pusty terminator oznacza brak wyraźnej separacji.
Listę niepustą mozemy wymusić np.

```
separator nonempty Ident "," ;
```

Gdy SLR(1) zawodzi...

Rozważmy gramatykę

$$
\begin{aligned}
S &\to L = R \mid R \\
L &\to * R \mid \mathbf{a} \\
R &\to L
\end{aligned}
$$

Próba budowy automatu SLR(1) doprowadzi nas do stanu

$$
\begin{aligned}
S &\to L \bullet = R \\
R &\to L \bullet
\end{aligned}
$$

W którym dla $=$ na wejściu możliwe jest zarówno przesunięcie jak i redukcja (do R). Okazuje się przy tym, że $= \in \text{FOLLOW}(R)$ i konfliktu nie da się usunąć metodą SLR(1).

Potrzebujemy silniejszego narzędzia.

Sytuacje LR(1)

Sytuacja LR(1)

$$[A \to \alpha \bullet \beta, a]$$

czyli para zawierająca sytuację LR(0) i terminal.
Jesteśmy w trakcie rozpoznawania $A \to \alpha\beta$,
na stosie jest już α, trzeba jeszcze rozpoznać β.

Sytuacja $[A \to \alpha \bullet, a]$ oznacza, ze na stosie mamy całą prawą stronę produkcji; możemy redukować gdy na wejściu jest a.

Notacja

$$[A \to \alpha \bullet X\beta, a, b, \ldots]$$

oznacza zbiór sytuacji

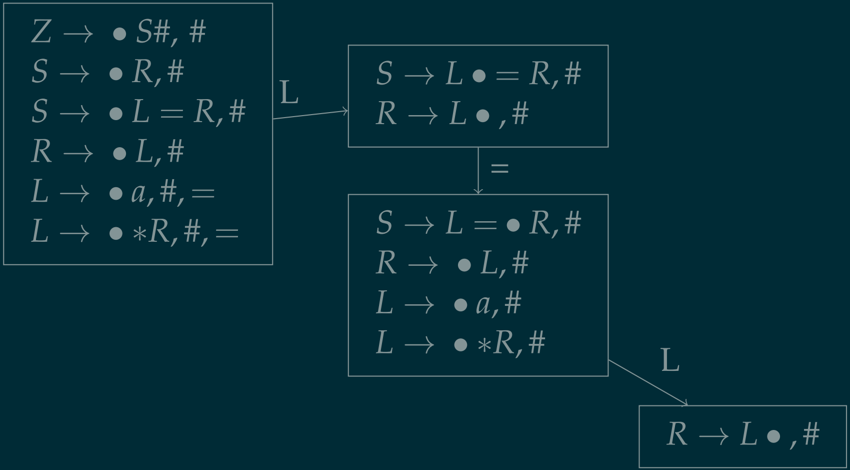
$$\{[A \to \alpha \bullet X\beta, a], [A \to \alpha \bullet X\beta, b], \ldots\}$$

Ponadto, jeśli nie powoduje to niejasności, opuszczamy nawiasy i piszemy

$$A \to \alpha \bullet X\beta, a, b, \ldots$$

Przykład (fragment automatu)

Jeśli mamy sytuację $[A \to \alpha \bullet X\beta, a]$, to po rozpoznaniu X mamy sytuację $[A \to \alpha X \bullet \beta, a]$



Stany i przejścia automatu LR(1)

- Stanami automatu są zbiory sytuacji LR(1).
- Jeśli jesteśmy w sytuacji $[B \to \alpha \bullet A\beta, a]$, to w wyprowadzeniu po A może wystąpić symbol z $\textsc{First}(\beta a)$. Jesteśmy zatem też w sytuacji $[A \to \bullet\gamma, b]$ dla każdego $A \to \gamma \in P$ oraz $b \in \textsc{First}(\beta a)$.
- Stan musi być domknięty zwn tę implikację:
- $Closure(Q)$ – najmniejszy zbiór zawierający Q oraz taki, że jeśli $[B \to \alpha \bullet A\beta, a] \in Closure(Q)$,to

$$\forall A \to \gamma \in P, b \in \textsc{First}(\beta a) \quad [A \to \ \bullet \gamma, b] \in Closure(Q)$$

- Jeśli $[A \to \alpha \bullet X\gamma, a] \in Q$ dla pewnego $X \in N \cup T$, to ze stanu Q jest przejście (po X) do stanu $Closure\left(\{[A \to \alpha X \bullet \gamma, a]\}\right)$.

Redukcje LR(1)

Jeśli stan p zawiera $[A \to \alpha \bullet, a]$, to: wpisujemy **reduce**$(A \to \alpha)$ do ACTION$[p, a]$

Jeśli gdzieś zostanie wpisana więcej niż jedna akcja, to źle: gramatyka nie jest klasy LR(1) (konflikt shift-reduce lub reduce-reduce).

W naszym przykładzie, w stanie

$$\boxed{\begin{array}{l} S \to L \bullet = R, \# \\ R \to L \bullet, \# \end{array}}$$

wpiszemy redukcję dla # a shift dla =. Nie ma konfliktu.

Usprawnianie metody LR(1)

W automacie LR(1) istnieje zwykle wiele podobnych stanów, np

$$[R \to L\bullet, \#, =]$$

oraz

$$[R \to L\bullet, \#]$$

Często możemy zmniejszyć automat, sklejając podobne stany.

Definicja

Jądro *zbioru sytuacji LR(1) to następujący biór sytuacji LR(0):*

$$kernel(p) = \{A \to \alpha \bullet \beta : \exists a \in T.\, [A \to \alpha \bullet \beta, a] \in p\}$$

Konstrukcja automatu LALR(1)

- Budujemy automat ze zbiorów sytuacji LR(1).
- Sklejamy równoważne stany (sumujemy stany mające identyczne jądra).
- Dalej postępujemy jak w metodzie LR(1).
- Jeśli nie powstaną nowe konflikty, to gramatyka jest LALR(1).

Zauważmy, że:

- Względem LR(1) mogą powstać tylko konflikty reduce-reduce, bo gdyby był konflikt shift-reduce, to istniałby i przy metodzie LR(1).
- automat LALR(1) ma tyle samo stanów co w metodzie LR(0)

Przykład na tablicy