Metody Realizacji Języków Programowania

Analiza składniowa wstępująca

Marcin Benke

MIM UW

8 listopada 2010

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.

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).

Definicja (uchwyt)

$A \to \beta$ jest *uchwytem* w prawostronnej formie zdaniowej $\alpha\beta w$, jeśli

$$S \to^R \alpha A w \to^* \alpha\beta w$$

dla pewnych $\alpha, \beta \in (N \cup T)*$, $w \in T^*$.

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 $k \leq 1$.

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}.

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

LR(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)

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 $A \to \alpha \bullet$ oznacza, ze na stosie mamy całą prawą stronę produkcji i możemy redukować (w metodzie SLR(1) tylko gdy na wejściu mamy $a \in \text{FOLLOW}(A)$).

Gdy SLR(1) zawodzi...

Rozważmy gramatykę

$$S \to L = R \mid R$$
$$L \to * R \mid \mathbf{a}$$
$$R \to L$$

Próba budowy automatu SLR(1) doprowadzi nas do stanu

$$S \to L\bullet = R$$
$$R \to L\bullet$$

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

Potrzebujemy silniejszego narzędzia.

Analiza problemu

Dlaczego $= \in \textsc{Follow}(R)$? Weźmy na przykład wyprowadzenie

$$S \to L = R \to *R = R \to \ldots$$

Tym niemniej nie istnieje wyprowadzenie prawostronne w tej gramatyce, które byłoby postaci

$$S \to^* \mu R = w \to \mu L = w \to \ldots$$

Zatem przy odtwarzaniu wyprowadzenia prawostronnego, jeśli następnym znakiem na wejściu jest “=”, to redukcja produkcji $R \to L$ nie doprowadzi nas do sukcesu.

Musimy sprawić, by nasz automat o tym “wiedział”.

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.
Ponadto istnieje wyprowadzenie prawostronne postaci

$$S \to^* \mu A a w \to \mu\alpha\beta a w \to \dots$$

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

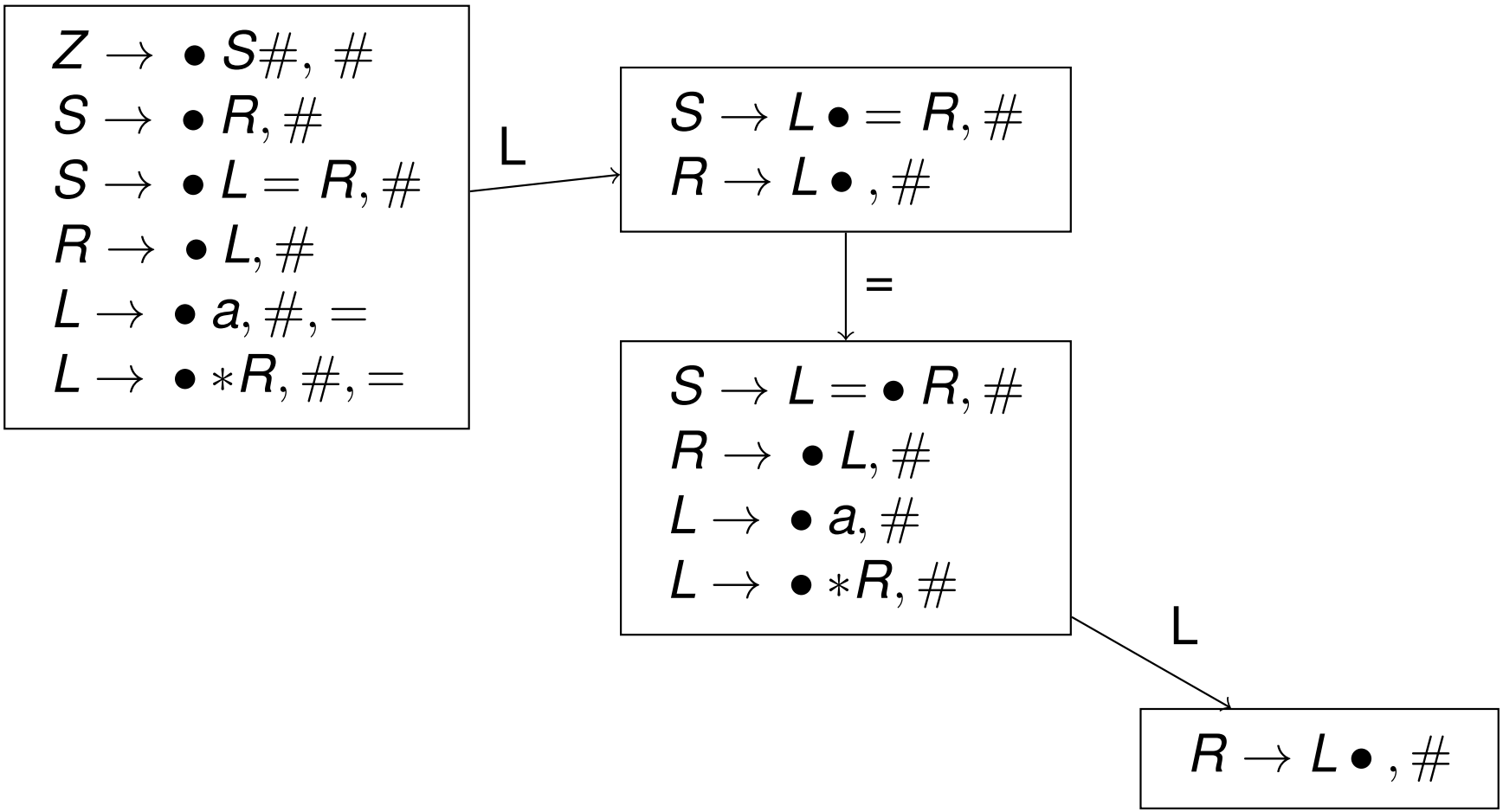
$$\{[A \to \alpha \bullet X\beta, a], [A \to \alpha \bullet X\beta, b], \ldots\}$$

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 $Closure(\{[A \to \alpha X \bullet \gamma, a]\})$.

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:
  - ACTION$[s, a]$ = **shift** p
    przenosi a z wejścia na stos i nakrywa stanem p
  - ACTION$[s, a]$ = **reduce**$(A \to \alpha)$
    zdejmuje $|\alpha|$ par ze stosu
    odsłoni się stan q (zawierał sytuację $\ldots \bullet A \ldots$)
    wkłada na stos A, 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 LR(1)
   - stan początkowy: $\mathit{Closure}(\{[Z \to \bullet\, S\#, \#]\})$
   - dla stanu p przejście po symbolu X do stanu

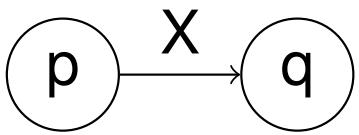
$$q = \mathit{Closure}([\{A \to \alpha X \bullet \gamma, a] : [A \to \alpha \bullet X\gamma, a] \in p\})$$

3. Wypełniamy tablicę sterującą automatu ze stosem.

Przykład na tablicy

Wypełnianie tablic sterujących

Numerujemy stany, numerujemy produkcje.
Wpisujemy akcje **shift** (przepisujemy przejścia automatu skończonego) i **accept**:

Dla przejścia 

wpisujemy:

- jeśli X jest *terminalem* to

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

- jeśli X jest *nieterminalem* to

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

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

Redukcje LR(1)

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

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

Przykład na tablicy

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

Dalsze usprawnienia

- Wchodzenie do stanu, w którym jest tylko jedna sytuacja typu $A \to \alpha \bullet$ nie ma sensu, bo tam zawsze redukujemy.
- Wprowadzamy nową akcję *shift-reduce j*: połączenie shift i reduce *j*.
- Usuwanie takich stanów — zysk rzędu 30%
- W tablicy moga występować miejsca nieosiągalne — nieistotne.
- Informację o błędach można przeunąć do osobnej tablicy *Err* — bitowej.
- Teraz możemy skleić wiersze różniące się tylko na miejscach pustych i nieistotnych.

Gramatyki niejednoznaczne

Rozważmy następującą (niejednoznaczną) gramatykę:

$$E \to E + E \mid E * E \mid a$$

Budując dla niej automat (np. LR(0)) natkniemy się na stan:

$$\begin{array}{l} E \to E + E \bullet \\ E \to E \bullet + E \\ E \to E \bullet * E \end{array}$$

Mamy tu konflikty shift-reduce dla $+$ i $*$ na wejściu. Jeżeli wybierzemy: dla $+$ reduce, a dla $*$ shift (i podobnie dla drugiego stanu z konfliktami) — uzyskamy tradycyjne priorytety operatorów.

Jak to sformalizować?

- Niektórym terminalom i produkcjom przypisujemy *priorytety*.
- Domyślnie produkcja otrzymuje priorytet ostatniego terminala.
- W sytuacji konfliktu: shift terminala kontra redukcja produkcji — wybieramy wyższy priorytet.
- Przy równych priorytetach wybieramy w/g kierunku wiązania operatora.

Przykład w Bisonie

```
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* leksem-widmo: minus unarny */
%right '^'    /* potęgowanie                */
%%
exp:      NUM                { $$ = $1;          }
        | exp '+' exp        { $$ = $1 + $3;     }
        | exp '-' exp        { $$ = $1 - $3;     }
        | exp '*' exp        { $$ = $1 * $3;     }
        | exp '/' exp        { $$ = $1 / $3;     }
        | '-' exp  %prec NEG { $$ = -$2;         }
        | exp '^' exp        { $$ = pow ($1,$3);}
        | '(' exp ')'        { $$ = $2;          }
;
%%
```

Generalized LR

Metoda oryginalnie wymyślona przez Tomitę dla analizy języków naturalnych.

- Budowa automatu LR dla gramatyki niejednoznacznej prowadzi do konfliktów (kilka możliwych akcji w jednej sytuacji).
- Każdy element tablicy automatu GLR moze zawierać zbiór akcji.
- Jeśli w danym momencie mamy zbiór akcji (> 1), automat *rozmnaża* się na odpowiednią liczbę kopii.
- Przy napotkaniu błędu kopia ginie
- Efekt: zbiór możliwych rozbiorów danego tekstu.
- Niektóre generatory (Bison,Happy) potrafią generować parsery GLR.

Obsługa błędów w metodzie LR

- Wprowadzamy specjalny leksem `error` — może być używany w produkcjach, np $E \rightarrow$ (`error`)
- Budujemy automat jak zwykle
- Przy napotkaniu błędu, automat zdejmuje ze stosu pary (stan,symbol) aż do napotkania stanu w którym jest akcja shift dla leksemu `error`
- Wykonujemy akcję shift dla `error`
- jeżeli kolejne leksemy nie są dopuszczalne w osiągniętym stanie, pomijamy aż do uzyskania dopuszczalnego.
- Kontynuujemy analizę

Przykład w Bisonie

```
exp:      NUM                { $$ = $1;          }
        | exp '+' exp        { $$ = $1 + $3;     }
        | exp '-' exp        { $$ = $1 - $3;     }
        | exp '*' exp        { $$ = $1 * $3;     }
        | exp '/' exp        { $$ = $1 / $3;     }
        | '-' exp  %prec NEG { $$ = -$2;         }
        | exp '^' exp        { $$ = pow ($1,$3);}
        | '(' exp ')'        { $$ = $2;          }
        | '(' error ')'
        { printf("Error in expression\n");$$ = 0;}
;
```

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 (wersje dla $\leq$ 1.4 i $\geq$ 1.5)
  - C
  - C++
  - O'Caml

Tu w skrócie, więcej — google bnf converter

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
ETimes. Exp ::= Term "*" Factor
...
```

Da zbyt rozgadaną składnię abstrakcyjną:

```
data Exp =  EPlus Exp Term | ETerm Term
data Term = TTimes Term Factor | TFact Factor
data Factor = EInt Integer
```

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 czystym 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 "," ;
```