Metody Realizacji Języków Programowania

Analiza składniowa zstępująca

Marcin Benke

MIM UW

5 grudnia 2016

Analiza syntaktyczna

- Analizator syntaktyczny (*parser*) jest funkcją sprawdzającą, czy dane słowo należy do języka i, jeśli tak, budującą drzewo struktury.
- Algorytm Youngera (CYK): pesymistycznie $\mathcal{O}(n^3)$.
- 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: znajdujemy możliwe konstrukcje; drzewo budowane od liści do korzenia ze znalezionych kawałków.

Analiza składniowa zstępująca (top-down)

Schemat analizy możemy 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

Powyższy 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 $SELECT(A \to \alpha)$.

Zbiory *FIRST*

Notacja

Niech $w \in T^*$

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

(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."

Zbiory *SELECT*

Definicja

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

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

Niech $FIRST'(\alpha) = FIRST(\alpha) \setminus \{\#\}$. Zauważmy, że

- jeśli $\alpha \to^* \epsilon$, to

$$SELECT(A \to \alpha) = FIRST'(\alpha) \cup FOLLOW(A)$$

- jeśli $\alpha \not\to^* \epsilon$

$$SELECT(A \to \alpha) = FIRST(\alpha) = FIRST'(\alpha)$$

Gramatyki silnie LL(k)

Definicja

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

Wniosek: gramatyka jest LL(1) wtw gdy dla każdej pary (różnych) produkcji $A \to \alpha$, $A \to \beta$ ich zbiory *SELECT* są rozłączne.

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

Pierwszy problem 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

$$E \to TR$$
$$R \to +TR \mid \varepsilon$$

Wyliczanie FIRST

Dla $t \in T$ mamy $FIRST(t) = \{t\}$.

Dla $A \in N$, $\quad A \to \alpha_1 \mid \ldots \mid \alpha_n$ mamy:

$FIRST(A) = FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n)$

Dla $A \to X_1 \ldots X_n$

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

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

Proste wyliczanie zbiorów FIRST

```
first :: Grammar -> Symbols -> Symbols
first g sy = maybeAddEot $ first' g sy where
  maybeAddEot ss | nullable g sy = EOT:ss
                 | otherwise = ss
first' :: Grammar -> Symbols -> Symbols
first' g sy = go [] sy where
 go _ []                             = []
 go v (h:t)
  | h `is_terminal` g = [h]
  | nullable_nt g h    = go (h:v) t ++ goNT v h
  | otherwise          = goNT v h
 goNT v h
  | h `elem` v = [] -- go v t
  | True = [s | nt <- rhs_nt g h,s<- go (h:v) nt]
rhs_nt :: Grammar -> Symbol -> [Symbols]
-- lista prawych stron produkcji dla danego symbolu
```

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 $FIRST'(\beta) \subseteq FOLLOW(X)$
- $A \to \alpha X \beta \in P$, $\beta \to^* \varepsilon$ 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)

$$E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to (E) \mid \mathbf{a}$$

Usuwamy lewostronną rekursję:

$$E \to TE' \qquad E' \to +TE' \mid \varepsilon \qquad T \to FT' \qquad T' \to *FT' \mid \varepsilon$$

$E \to TE'$	niepotrzebne $\textsc{Select}$
$E' \to +TE'$ $E' \to \varepsilon$	$\textsc{Select}(E' \to +TE') = \{+\}$ $\textsc{Select}(E' \to \varepsilon) = \textsc{Follow}(E') = \{), \#\}$
$T \to FT'$	
$T' \to *FT'$ $T' \to \varepsilon$	$\textsc{Select}(T' \to *FT') = \{*\}$ $\textsc{Select}(T' \to \varepsilon) = \textsc{Follow}(T') = \{+,), \#\}$
$F \to (E)$ $F \to \mathbf{a}$	$\textsc{Select}(F \to (E)) = \{(\}$ $\textsc{Select}(F \to \mathbf{a}) = \{\mathbf{a}\}$

Metoda zejść rekurencyjnych

Metoda tworzenia parsera top-down jako zbioru wzajemnie rekurencyjnych funkcji:

1. przekształcamy gramatykę do postaci LL(1)
2. liczymy zbiory SELECT
3. (wersja ortodoksyjna)
   dla każdego nieterminala A piszemy osobną, rekurencyjną funkcję A.

Funkcja A rozpoznaje najdłuższy ciąg terminali (leksemów) wyprowadzalny z A.

Styk z analizatorem leksykalnym

Niezmiennik

Zawsze mamy jeden nie zużyty leksem;
Funkcja startując ma już wczytany leksem, po jej zakończeniu na wejściu jest pierwszy leksem nie należący do ciągu wyprowadzonego z A.

Zachowanie niezmiennika jest kluczowe dla poprawności.

Styk z analizatorem leksykalnym:

- bieżący leksem (tu: `lexem`)
- funkcja pobierająca następny leksem (tu: `next()`)

Przy starcie analizatora musimy mieć gotowy pierwszy leksem

Ogólny schemat

```
void A() {
  switch(lexem) {
    case L: // dla L w SELECT(A->X1...Xk)
      dla kolejnych Xi wykonuj:
        jeśli Xi jest terminalem:
          if(lexem==Xi) next();
          else błąd: oczekiwano Xi;
        jeśli Xi jest nieterminalem:
          Xi();
      break;
    case ...
    default:
      błąd: oczekiwano jednego z: ...
  }
}
```

Przykład

```
void expect(Lexem l) {
   if(l==lexem) next(); else błąd ...
}
void E(){ // E -> T E1
  T(); E1();
}
void F() { // F -> (E) | num
  switch(lexem) {
    case lewias:
      next(); E(); expect(prawias); break;
    case num: next(); break;
    default: błąd
  }
}
```

Przykład c.d.

```
void E1() { // E1 -> + T E1 | epsilon
  if (lexem == plus) {
    next(); T(); E1();
}
```

Jeśli lexem $\neq$ plus oraz $\mathrm{lexem} \notin \{\mathrm{prawias}, \mathrm{koniec}\}$ czyli SELECT($E1 \to \varepsilon$), to już wiadomo, że błąd.

Ale dla ε-produkcji byłaby to nadgorliwość; błąd i tak zostanie wykryty w tym samym miejscu przez funkcję oczekującą konkretnych terminali.

Wersja pragmatyczna

Zakładając, że mamy już gramatykę LL(1) i policzone zbiory SELECT:

1. dla każdego nieterminala tworzymy graf składniowy; rozgałęzienie odpowiada wyborowi produkcji, zatem zbiory SELECT służą wyborowi drogi.
2. Sklejamy grafy, aby zmniejszyć ich liczbę, a przez to i liczbę wywołań funkcji.
3. Zastępujemy rekursję ogonową przez iterację.
4. Dla każdego grafu piszemy funkcję; graf jest schematem blokowym takiej funkcji i wystarczy go starannie zakodować.

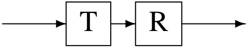
Przykład

Gramatyka:

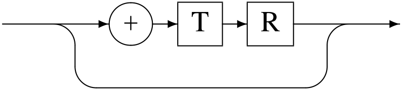
$$E \to TR \qquad R \to +TR \mid \varepsilon$$

Grafy składniowe:

E

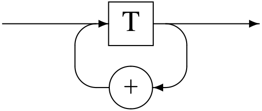


R



Po połączeniu grafów i zastąpieniu rekursji ogonowej iteracją:

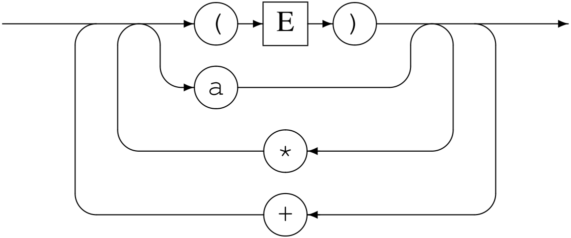
E




```
for(int stop=0;!stop;) {
  T();
  if(lexem==PLUS)
    nextLexem();
  else
    stop=1;
  }
}
```

Po sklejeniu wszystkich diagramów:

E



Obsługa błędów

Fakt

W metodzie LL(1) błąd zostanie wykryty dla pierwszego symbolu a t.że (o ile wcześniej wczytano α), αa nie jest prefiksem żadnego słowa z $L(G)$.

- Znamy jedynie **miejsce wykrycia** i **objawy** błędu a nie sam błąd
- Każdy sposób obsługi błędu może spowodować **lawinę** (pozornych) błędów.

Jak kontynuować analizę

1. Znaleźć możliwie małe poddrzewo zawierające błąd.
2. Pominąć leksemy aż do końca tego poddrzewa (czyli do napotkania leksemu ze zbioru FOLLOW.

Na przykład:

```
void F() { // F -> (E) | num
  switch(lexem) {
    case lewias:
      next(); E(); expect(prawias); break;
    case num: next(); break;
    default:
      błąd(...);
      do {next();} while(!inFollowF(lexem));
  }
}
```

Budowa drzewa struktury

Zwykle dla wyjściowej gramatyki budowa drzewa struktury jest prosta: funkcje odpowiadające nieterminalom dają w wyniku węzeł odpowiedniego typu

$E \to E + T$	`BinOp('+', E(), T())`
$E \to T$	`T()`
$T \to T * F$	`BinOp('*', T(), F())`
$F \to \textbf{num}$	`Num(numvalue)`
$F \to (E)$	`E()`

Budowa drzewa struktury a transformacje LL(1)

Faktoryzacja gramatyki:

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

daje w wyniku:

$$\begin{aligned} E &\to TR \\ R &\to +E \mid \varepsilon \end{aligned}$$

Jak zbudować drzewo dla R?
Jakiego w ogóle typu ma być funkcja dla R?

Kontynuacje na pomoc

Możemy zauważyć, że R jest **kontynuacją** T. Argumentem dla R będzie węzeł zbudowany przez T:

```
Exp E() {
  Exp e = T();
  return R(e);
}
Exp R(Exp e){
  switch(lexem) {
    case PLUS: return BinOp('+',e, E());
    case ...: return e;
    ...
}
```

Eliminacja lewostronnej rekursji

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

staje się

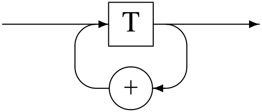
$$E \to TR$$
$$R \to +TR \mid \varepsilon$$

Czyli podobnie jak w poprzednim przypadku. Musimy tylko zadbać o zachowanie wiązania w lewo przy kodowaniu drugiej reguły:

```
Exp R(Exp e){
  switch(lexem) {
    case PLUS: return R(BinOp('+',e, T());
    case ...: return e;
    ...
}
```

Budowa drzewa w wersji “pragmatycznej”

E




```
Exp E() {
  Exp e = T();
  while(lexem==PLUS) {
    nextLexem();
    e = BinOp('+',e,T());
  }
  return e;
}
```

Procedury dla operatorów wiążących w prawo można pozostawić w wersji rekurencyjnej (czyli tak jak w wersji “ortodoksyjnej”).

Analiza zstępująca w Haskellu

W Haskellu mozna programować imperatywnie, mając odpowiednią monadę:

```
class MonadError String m => LPM m where
  type Token m
  runLPM :: m a -> [Token m]
         -> Either String (a,[Token m])
  getLex :: m (Token m)
  nextLex :: m ()

expect c = do
  c' <- getLex
  if c == c' then return ()
             else throwError $ unwords
                   ["expected",[c],"got",[c']]

instance LPM Parser where ...
```

Zejścia rekurencyjne w Haskellu

Wtedy dla fragmentu gramatyki

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

możemy napisać

```
pE :: P Exp
pE = pT >>= pE'

pE' :: Exp -> P Exp
pE' t = getLex >>= go where
  go '+' = do
    nextLex
    t2 <- pT
    pE' $ EAdd t t2
  go _ = return t
```

. . . ale może lepiej wykorzystać mozliwości jakie daje język funkcyjny.

Kombinatory parsujące

Kombinatory (funkcje) reprezentujące elementarne parsery i sposoby łączenia parserów:

```
item :: Parser Char
(<|>) :: Parser a -> Parser a -> Parser a
satisfy :: (Char->Bool) -> Parser Char
char :: Char -> Parser Char
char x = satisfy (==x)
digit :: Parser Int
many, many1 :: Parser a -> Parser [a]
```

Sekwencjonowanie zwykle przy użyciu monad, np

```
do { m <- digit; n <- digit; return 10*m+n}
```

Nie wszystkie biblioteki używają monad, niektóre są oparte na `Applicative`

Kombinatory parsujące

Przy użyciu tych kombinatorów mozemy np. dla gramatyki

```
Int :: Nat | '-' Nat
Nat ::= {digit}
```

Napisać parser(y):

```
pInt, pNat :: Parser Integer
pInt =  pNat  <|> negative pNat  where
  negative :: (Num a) => Parser a -> Parser a
  negative p = fmap negate (char '-' >> p)

pNat = fmap (foldl (\x y -> 10*x+toInteger y) 0)
            pDigits
```

Ograniczanie niedeterminizmu

Ponieważ pełny niedeterminizm może prowadzić do wykładniczej eksplozji złożoności, operator wyboru $<|>$ można uczynić deterministycznym (zadziała dla gramatyk LL(1)).

```
many, many1 :: Parser a -> Parser [a]
many p = many1 p <|> return []

many1 p = do { a <- p; as <- many p; return (a:as)}
```

Zauważmy, że kolejność argumentów operatora wyboru ma teraz znaczenie i **many** tak jak jest zdefiniowane da nam *najdłuższe* możliwe dopasowanie (przewaznie tego właśnie chcemy).

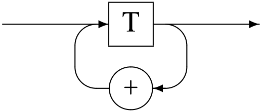
Parsec

Pakiet **parsec** (zainstalowany na students) dostarcza bibliotekę kombinatorów parsujących

```
import Text.ParserCombinators.Parsec

-- type Parser a = GenParser Char () a
-- Defined in Text.ParserCombinators.Parsec.Prim
-- parse :: GenParser tok () a -> SourceName
--         -> [tok] -> Either ParseError a

run :: Parser a -> [Char] -> Either ParseError a
run p s = parse p "(interactive)" s

pInt :: Parser Integer
pInt = negative pNat <|> pNat where
  negative :: (Num a) => Parser a -> Parser a
  negative p = fmap negate (char '-' >> p)
...
```

Parsec

E



Zamiast pętli Parsec oferuje kombinatory `chainl1, chainr1`

```
pExp, pT, pF :: Parser Exp
pExp = pT `chainl1` addop
pT = pF `chainl1` mulop
pF = ENum <$> integer <|> parens pExp
addop =  do{ symbol "+"; return EAdd }
          <|> do{ symbol "-"; return ESub }

data Exp = ENum Integer
         | EAdd Exp Exp | ESub Exp Exp
         | EMul Exp Exp | EDiv Exp Exp
```