Metody Realizacji Języków Programowania

Analiza składniowa II

Marcin Benke

MIM UW

18 października 2010

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:

Po połączeniu grafów i zastąpieniu rekursji ogonowej iteracją:

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

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())`
$F \to$ **num**	`Num(numvalue)`
$F \to (E)$	`E()`

Budowa drzewa struktury a transformacje LL(1)

Faktoryzacja gramatyki:

$$E \rightarrow T + E \mid T$$

daje w wyniku:

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

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

$$E \rightarrow E + T \mid T$$

staje się

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

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”

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

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

Translacja sterowana składnią

Projektujemy kompilator starając się myśleć tylko o jednej konstrukcji języka naraz.

Wszelkie potrzebne informacje i czynności chcemy rozdzielić na związane z osobnymi, pojedynczymi konstrukcjami

Jednym z narzędzi do tego służących są *gramatyki atrybutywne*

Akcje i atrybuty

Dla każdej konstrukcji z osobna definiujemy:

- jakie informacje są potrzebne (atrybuty)
- od jakich innych informacji zależą (jak je wyliczyć)
- jakie czynności trzeba wykonać (akcje)

Gramatyka translacyjna

{akcja} — nowy rodzaj symbolu,

- może występować tylko po prawej stronie produkcji,
- może używać i wyliczać wartości atrybutów

Akcje są wykonywane w kolejności występowania w wyprowadzeniu lewostronnym.

Przykład

Kod maszyny stosowej dla wyrażeń arytmetycznych

$E \rightarrow E + T$ {Add}
$E \rightarrow T$
$T \rightarrow T * F$ {Mul}
$T \rightarrow F$
$F \rightarrow \mathrm{num}$ {Push num.value}
$F \rightarrow (E)$

Wyprowadzenie $1 + 2 * 3$
E
E+T{Add}
T+T{Add}
F+T{Add}
1{Push 1}+**T** {Add}
1{Push 1}+**T***F{Mul} {Add}
1{Push 1}+**F***F{Mul} {Add}
1{Push 1}+2{Push 2}***F**{Mul} {Add}
1{Push 1}+2{Push 2}*3{Push 3} {Mul} {Add}

Wybierając same akcje otrzymujemy:
{Push 1} {Push 2} {Push 3} {Mul} {Add}

Gramatyki atrybutywne

Gramatyka atrybutywna

$$AG = \langle G, A, R \rangle$$

G — gramatyka bezkontekstowa, A — zbiór atrybutów, R — zbiór reguł atrubutowania

Niech A(X) — zbiór atrubutów symbolu X;
X.a oznacza atrybut a symbolu X.
Dla produkcji $p : X_0 \to X_1 \ldots X_n$ definiujemy reguły atrybutowania

$$R(p) = \{X_i.a \leftarrow f_{i,a}(X_j.b \ldots X_k.c) \mid 0 \leq i \leq n, a \in A(X_i)\}$$

Well defined Attribute Grammar

Mając drzewo struktury chcemy dla każdego wierzchołka X wyznaczyć wartości wszystkich atrubutów zgodnie z regułami atrybutowania.

Definicja (WAG)

Gramatyka atrybutywna jest **dobrze zdefiniowana** jeśli dla każdego drzewa struktury zgodnego z tą gramatyką można w sposób jednoznaczny wyznaczyć wartości wszystkich atrybutów.

Nieważne “jak”, ważne, że “można”.
Zagrożenia: brak reguły, sprzeczne reguły, cykl

Atrybuty syntetyzowane i dziedziczone

Dla każdej produkcji $p : X_0 \to X_1 \ldots X_n$ zbiorem **definiujących wystapień atrybutów** jest

$$AF(p) = \{X_i.a \mid X_i.a \leftarrow f(\cdots) \in R(p)\}$$

- Atrybut $X.a$ jest **syntetyzowany**, jeśli istnieje produkcja $p : X \to \alpha$ i $X.a \in AF(p)$ (czyli zależy od poddrzewa)
- Atrybut $X.a$ jest **dziedziczony**, jeśli istnieje produkcja $q : Y \to \alpha X \beta$ i $X.a \in AF(q)$ (czyli zależy od otoczenia)

Oznaczenia:
$AS(X)$ — atrybuty syntetyzowane X,
$AI(X)$ — atrybuty dziedziczone X.
Dla symboli terminalnych mówimy o **atrybutach wbudowanych** — dane przez lekser.

Gramatyki zupełne

Gramatyka jest zupełna, jeśli dla każdego symbolu X spełnione są warunki:

1. dla każdej produkcji $p : X \to \alpha$ mamy $AS(X) \subseteq AF(p)$,
2. dla każdej produkcji $q : Y \to \alpha X \beta$ mamy $AI(X) \subseteq AF(p)$,
3. $AS(X) \cup AI(X) = A(X)$,
4. $AS(X) \cap AI(X) = \emptyset$.

Każda gramatyka zupełna jest dobrze zdefiniowana. Możliwa implementacja:

- wierzchołki drzewa struktury — obiekty odp. klas
- atrybuty syntetyzowane — metody wirtualne
- atrubuty dziedziczone — przekazywane jako argumenty tychże metod,

Atrybuty można przechowywać także jako atrybuty wierzchołków, by uniknąć wielokrotnego ich wyliczania.

Przykład — atrybut syntetyzowany

Konwencja: jeśli dany symbol występuje więcej niż raz w danej produkcji, jego wystąpienia numerujemy.

Atrybuty: $E.val$, $T.val$, $F.val$ — syntetyzowane, $num.val$ — wbudowany

$E_0 \rightarrow E_1 + T \ \{E_0.val \leftarrow E_1.val + T.val\}$
$E \rightarrow T \ \{E.val \leftarrow T.val\}$
$T \rightarrow T * F\{T_0.val \leftarrow T_1.val * F.val\}$
$T \rightarrow F \ \{T.val \leftarrow F.val\}$
$F \rightarrow num \ \{F.val \leftarrow num.val\}$
$F \rightarrow (E) \ \{F.val \leftarrow E.val\}$

Przykład — atrybut dziedziczony

$D \rightarrow TL\ \{L.typ \leftarrow D.typ;\ D.typ \leftarrow T.typ\}$
$T \rightarrow \mathrm{int}\ \{T.typ \leftarrow \mathrm{int}\}$
$T \rightarrow \mathrm{real}\ \{T.typ \leftarrow \mathrm{real}\}$
$L_0 \rightarrow L_1, \mathrm{id}\ \{L_1.typ \leftarrow L_0.typ,\ id.typ \leftarrow L_0.typ\}$ $L \rightarrow \mathrm{id}\ \{id.typ \leftarrow L.typ\}$

Atrybuty:

- $T.typ$, $D.typ$ — syntetyzowany
- $L.typ$ — dziedziczony
- $id.typ$ — dziedziczony