Syntaktická analýza

Implementace $LL(1)$ překladů

Šárka Vavrečková

Ústav informatiky, FPF SU Opava

sarka.vavreckova@fpf.slu.cz

Poslední aktualizace: 27. října 2023

Postup

Programujeme syntaktickou analýzu:

1. Navrhneme vhodnou $LL(1)$ gramatiku popisující strukturu jazyka.
2. Vytvoříme podle ní překladový automat (rozkladovou tabulku) nebo alespoň všechny potřebné množiny.
3. Naprogramujeme:
   - metodou přepisu rozkladové tabulky,
   - metodou rekurzivního sestupu.

Postup

Programujeme syntaktickou analýzu:

1. Navrhneme vhodnou $LL(1)$ gramatiku popisující strukturu jazyka.
2. Vytvoříme podle ní překladový automat (rozkladovou tabulku) nebo alespoň všechny potřebné množiny.
3. Naprogramujeme:
   - metodou přepisu rozkladové tabulky,
   - metodou rekurzivního sestupu.

Postup

Programujeme syntaktickou analýzu:

1. Navrhneme vhodnou $LL(1)$ gramatiku popisující strukturu jazyka.
2. Vytvoříme podle ní překladový automat (rozkladovou tabulku) nebo alespoň všechny potřebné množiny.
3. Naprogramujeme:
   - metodou přepisu rozkladové tabulky,
   - metodou rekurzivního sestupu.

Postup

Programujeme syntaktickou analýzu:

1. Navrhneme vhodnou $LL(1)$ gramatiku popisující strukturu jazyka.
2. Vytvoříme podle ní překladový automat (rozkladovou tabulku) nebo alespoň všechny potřebné množiny.
3. Naprogramujeme:
   - metodou přepisu rozkladové tabulky,
   - metodou rekurzivního sestupu.

Postup

Programujeme syntaktickou analýzu:

1. Navrhneme vhodnou $LL(1)$ gramatiku popisující strukturu jazyka.
2. Vytvoříme podle ní překladový automat (rozkladovou tabulku) nebo alespoň všechny potřebné množiny.
3. Naprogramujeme:
   - metodou přepisu rozkladové tabulky,
   - metodou rekurzivního sestupu.

Přepis rozkladové tabulky

Potřebujeme

- rozkladovou tabulku,
- zásobník na ukládání symbolů,
- proměnnou, ve které je uložen právě zpracovávaný symbol,
- funkci `lex()`, která nám vrátí další symbol, který extrahovala ze vstupního souboru (uloží do proměnné z předchozího bodu),
- proměnnou pro výstup (soubor, dynamická struktura apod.).

Popis metody

Budeme potřebovat tyto funkce:

- `expand(číslo_pravidla)` uloží pravou stranu pravidla s daným číslem do zásobníku a na výstup přidá číslo pravidla,
- `pop()` ověří shodnost symbolu na vstupu se symbolem ze zásobníku a posune se na vstupu,
- `accept()` při konci vstupu a konci zásobníku ukončí program,
- `error()` ošetří chybu, která se vyskytla při překladu,
- `Akce()` je hlavní řídicí funkce, v cyklu volá předchozí, zajišťuje „pohyb v tabulce“,
- `Init()`, `Done()` je inicializační a úklidová funkce.

Příklad

$S \to AB$	(1)
$A \to CD$	(2)
$B \to +AB \mid -AB \mid \varepsilon$	(3), (4), (5)
$C \to (S) \mid i \mid n$	(6), (7), (8)
$D \to *CD \mid /CD \mid \varepsilon$	(9), (10), (11)

Rozkladová tabulka

	i	n	$+$	$-$	$*$	$/$	$($	$)$	$\$$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

Příklad

$S \to AB$	①
$A \to CD$	②
$B \to +AB \mid -AB \mid \varepsilon$	③, ④, ⑤
$C \to (S) \mid i \mid n$	⑥, ⑦, ⑧
$D \to *CD \mid /CD \mid \varepsilon$	⑨, ⑩, ⑪

Rozkladová tabulka

	i	n	$+$	$-$	$*$	$/$	$($	$)$	\$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

Datové typy

```
enum TTypSymbolu { S_NOTHING, S_ENDOFFILE, S_LPAR, S_RPAR,
  S_ID, S_NUM, S_IS, S_PLUS, S_MINUS, S_MUL, S_DIV,     // terminály
   S_NS, S_NA, S_NB, S_NC, S_ND, S_HASH };       // neterminály
```

```
struct TSymbol {
  TTypSymbolu typ;
  string atrib;
};
```

```
struct TVstup {
  char znak;
  int cisloRad;
  int pozice;
  int konec;
};
```

```
TSymbol symbol;
TVstup vstup;
bool konec;
TZasobnik zasobnik;
TTypSymbolu vrchol_zas;
```

expanze pravidla

```
int expand(int cislo_prav) {
  switch (cislo_prav) {
  case  1:                        // S → AB
      Pridej_do_zasobniku(S_NB);
      Pridej_do_zasobniku(S_NA);
      break;
  case  2:                        // A → CD
      Pridej_do_zasobniku(S_ND);
      Pridej_do_zasobniku(S_NC);
      break;
  case  3:                        // B → +AB
      Pridej_do_zasobniku(S_NB);
      Pridej_do_zasobniku(S_NA);
      Pridej_do_zasobniku(S_PLUS);
      break;
    ... // pro každé pravidlo gramatiky
  };
  vystup(cislo_prav);           // zápis čísla použitého pravidla na výstup
}
```

Ošetření chyb

```
void error(string hlaska) {
// Chyba syntaxe;
// vypíše číslo řádku, pozici na řádku a řetězec s daným hlášením
  konec = true;
  printf("Chyba při syntaktické analýze na řádku %d, sloupci %d: %s",
    vstup.cisloRad, vstup.pozice, hlaska);
}
```

Zpracování terminálů a akceptování

```
int pop() {
  if (symbol.typ == vrchol_zas)
    Lex();                        // lexikální analyzátor načte další symbol
  else error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
}
```

```
void accept() {
  konec = true;
}
```

Inicializace a ukončení

```
void Init() {
  ...                          // inicializace vstupu a výstupu
  Vytvor_zasobnik();
  Pridej_do_zasobniku(S_HASH); // symbol konce zásobníku
  Pridej_do_zasobniku(S_NS);   // startovací symbol gramatiky
  Lex();                       // načte symbol ze vstupu do symbol
  konec = false;
}
```

```
void Done() {
  Zlikviduj_zasobnik();        // uvolní paměť zabranou zásobníkem
  ...                          // uzavření vstupu a výstupu
}
```

Simulace práce s tabulkou

Funkce `Akce`

Pracuje takto:

- vyjme ze zásobníku jeden symbol, tím určí řádek tabulky a podle symbolu na vstupu určí sloupec tabulky,
- podle obsahu buňky zavolá funkci `expand`, `pop`, `accept` nebo `error`,
- je volána v cyklu tak dlouho, dokud není konec zpracovávaného programu.

Simulace práce s tabulkou

```
switch (řádek) {
  case první_řádek: switch (sloupec) {
                case první_sloupec: obsah_buňky_[1,1]; break;
                case druhý_sloupec: obsah_buňky_[1,2]; break;
                ...
                default: error(...);
              } break;
  case druhý_řádek: switch (sloupec) {
                case první_sloupec: obsah_buňky_[2,1]; break;
                case druhý_sloupec: obsah_buňky_[2,2]; break;
                ...
                default: error(...);
              } break;
  ...
  case terminál: pop; break;
  case dno_zásobníku: if (konec_vstupu) accept(); else error(...);
}
```

Simulace práce s tabulkou

	i	n	$+$	$-$	$*$	$/$	$($	$)$	\$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

```
void Akce() {
  vrchol_zas = Vyjmi_ze_zasobniku();
  switch (vrchol_zas) {
    case S_NS: switch(symbol.typ) {
        case S_ID:
        case S_NUM:
        case S_LPAR: expand(1); break;
        default: error("chybný symbol na vstupu -" +symbol.typ);
    }  break;

    case S_NA: switch(symbol.typ) {
        case S_ID:
        case S_NUM:
        case S_LPAR: expand(2); break;
        default: error("chybný symbol na vstupu -" +symbol.typ);
    }  break;
  ...
```

Simulace práce s tabulkou

	i	n	$+$	$-$	$*$	$/$	$($	$)$	$\$$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

```
...
 case S_NB: switch (symbol.typ) {
   case S_PLUS: expand(3); break;
   case S_MINUS: expand(4); break;
   case S_RPAR:
   case S_ENDOFFILE: expand(5); break;
   default: error("chybný symbol na vstupu -" +symbol.typ);
 }  break;

 case S_NC: switch (symbol.typ) {
   case S_ID: expand(7); break;
   case S_NUM: expand(8); break;
   case S_LPAR: expand(6); break;
   default: error("chybný symbol na vstupu -" +symbol.typ);
 }  break;
...
```

Simulace práce s tabulkou

	i	n	$+$	$-$	$*$	$/$	$($	$)$	\$
S	e1	e1					e1		
A	e2	e2					e2		
B			e3	e4				e5	e5
C	e7	e8					e6		
D			e11	e11	e9	e10		e11	e11

```
    ...
     case S_ND: switch (symbol.typ) {
       case S_MUL: expand(9); break;
       case S_DIV: expand(10); break;
       case S_PLUS: case S_MINUS: case S_RPAR: case S_ENFOFFILE: expand(11); break;
       default: error("chybný symbol na vstupu -" +symbol.typ);
     }  break;

     case S_ID: case S_NUM: ... case S_RZAV: pop(); break;

     case S_HASH: if (symbol.typ == S_ENDOFFILE) accept();
       else error("chybný symbol na vstupu -" +symbol.typ);
       break;

     default: error("chybný symbol na vstupu -" +symbol.typ);
   }
}
```

Hlavní funkce syntaktické analýzy

```
void S_analyza() {
  Init();
  while (! konec)
    Akce();
  Done();
}
```

Vlastnosti metody

Výhody:

- nepoužíváme přímo rekurzi (netřeba řešit problém hloubky rekurze s prostorovou složitostí).

Nevýhody:

- u překladů zahrnujících např. matematické výrazy se hůře implementuje sémantika,
- potřebujeme zásobník.

Vlastnosti metody

Výhody:

- nepoužíváme přímo rekurzi (netřeba řešit problém hloubky rekurze s prostorovou složitostí).

Nevýhody:

- u překladů zahrnujících např. matematické výrazy se hůře implementuje sémantika,
- potřebujeme zásobník.

Rekurzívní sestup

Potřebujeme

- $LL(1)$ gramatiku (nemusíme dělat rozkladovou tabulku),
- množiny $FIRST$ a $FOLLOW$, pro každé pravidlo $A \to \alpha$ vytvoříme „množinu signatur“
$$FS(A, \alpha) = FIRST(\alpha \cdot FOLLOW(A))$$
- proměnnou, ve které je uložen právě zpracovávaný symbol,
- funkci `lex()`, která nám vrátí další symbol, který extrahovala ze vstupního souboru (uloží do proměnné z předchozího bodu),
- proměnnou pro výstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzivně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzivní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzívně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzívní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzívně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzívní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzívně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzívní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzívně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzívní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzívně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzívní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Analýza probíhá takto:

1. zavoláme funkci `lex()`, pak opět voláme pro každý symbol,
2. postupujeme tak, jakobychom konstruovali derivační strom „ručně“:
   - když jsme v uzlu ohodnoceném neterminálem, vytvoříme poduzly podle zvoleného pravidla,
   - tentýž postup rekurzívně na všechny potomky, kteří jsou ohodnoceni neterminály,
   - u terminálních poduzlů pouze spustíme „kontrolní porovnání“.
3. rekurzívní volání probíhá zleva doprava a shora dolů, vstup je čten zleva doprava,
4. když skončí rekurze a vstup je přečtený ($), akceptujeme vstup.

Popis metody

Budeme potřebovat tyto funkce:

- `Init()`, `Done()`,
- `pop()` ověří shodnost symbolů a posune se na vstupu,
- `S()`, `A()`, `B()`, ... – pro každý neterminál, tyto funkce se budou navzájem rekurzívně volat,
- `error()` ošetří chybu, která se vyskytla při překladu.

Příklad

$S \to AB$	(1)
$A \to CD$	(2)
$B \to +AB \mid -AB \mid \varepsilon$	(3), (4), (5)
$C \to (S) \mid i \mid n$	(6), (7), (8)
$D \to *CD \mid /CD \mid \varepsilon$	(9), (10), (11)

Datové typy

```
enum TTypSymbolu { S_NOTHING, S_ENDOFFILE, S_LPAR, S_RPAR,
  S_ID, S_NUM, S_IS, S_PLUS, S_MINUS, S_MUL, S_DIV };
```

```
struct TSymbol {
  TTypSymbolu typ;
  string atrib;
};
```

```
struct TVstup {
  char znak;
  int cisloRad;
  int pozice;
  int konec;
};
```

```
TSymbol symbol;
TVstup vstup;
bool konec;
```

Hlavní funkce syntaktické analýzy

Úkol:

- inicializovat výpočet,
- zavolat funkci S(), dále je vše voláno rekurzí,
- ukončit výpočet.

```
void S_analyza() {
  Init();
  S();
  Done();
}
```

Hlavní funkce syntaktické analýzy

Úkol:

- inicializovat výpočet,
- zavolat funkci S(), dále je vše voláno rekurzí,
- ukončit výpočet.

```
void S_analyza() {
  Init();
  S();
  Done();
}
```

Inicializace a ukončení

```
void Init() {
  ...                          // inicializace vstupu a výstupu
  Lex();                       // načte symbol ze vstupu do sym
  konec = false;
};


void Done() {
  ...                         // uzavření vstupu a výstupu
}
```

Ošetření chyb

```
// vypíše číslo řádku, pozici na řádku a řetězec s daným hlášením
void error(string hlaska) {
  konec = true;
  printf("Chyba při syntaktické analýze na řádku %d, sloupci %d: %s",
    vstup.cisloRad, vstup.pozice, hlaska);
}
```

Zpracování terminálů

- porovná zpracovávaný symbol (terminál z pravidla) se znakem na vstupu,
- načte další znak ze vstupu.

```
int pop(TTypSymbolu terminal) {
  if (symbol.typ == terminal)
    Lex();                      // lexikální analyzátor načte další symbol
  else error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
}
```

Funkce neterminálů

Pro každou množinu pravidel se stejnou levou stranou:

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

```
void A() {
  if (vstupni_sym) in FS(A,α1)
    ... postupně jsou ošetřeny symboly z řetězce α1
  else if (vstupni_sym) in FS(A,α2)
    ... postupně jsou ošetřeny symboly z řetězce α2
  else ... ostatní pravidla
  else error(...);
}
```

Funkce neterminálů

Pro každou množinu pravidel se stejnou levou stranou:

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

```
void A() {
  if (vstupni_sym) in FS(A,α1)
    ... postupně jsou ošetřeny symboly z řetězce α1
  else if (vstupni_sym) in FS(A,α2)
    ... postupně jsou ošetřeny symboly z řetězce α2
  else ... ostatní pravidla
  else error(...);
}
```

Funkce neterminálů

$S \rightarrow AB$

```
void S() {
  if (symbol.typ == S_ID || symbol.typ == S_NUM || symbol.typ == S_LPAR) {
    A();
    B();
  } else error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
}
```

Funkce neterminálů

$$A \rightarrow CD$$

```
void A() {
  if symbol.typ (symbol.typ == S_ID || symbol.typ == S_NUM || symbol.typ == S_LPAR) {
    C();
    D();
  } else error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
}
```

Funkce neterminálů

$$B \rightarrow +AB \mid -AB \mid \varepsilon$$

```
void B() {
  switch (symbol.typ) {
    case S_PLUS:
      pop(S_PLUS);
      A();
      B();
      break;
    case S_MINUS:
      pop(S_MINUS);
      A();
      B();
      break;
    case S_RPAR: case S_ENDOFFILE:  ;
    default: error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
  }
}
```

Funkce neterminálů

$$C \to (S) \mid i \mid n$$

```
void C() {
  switch (symbol.typ) {
    case S_LPAR:
      pop(S_LPAR);
      S();
      pop(S_RPAR);
      break;
    case S_ID:  pop(S_ID);  break;
    case S_NUM: pop(S_NUM); break;
    default: error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
  }
}
```

Funkce neterminálů

$$D \rightarrow *CD \mid /CD \mid \varepsilon$$

```
void D() {
  switch (symbol.typ) {
    case S_MUL:
      pop(S_MUL);
      C();
      D();
      break;
    case S_DIV:
      pop(S_DIV);
      C();
      D();
      break;
    case S_PLUS: case S_MINUS: case S_RPAR: case S_ENDOFFILE:  ;
    default: error("chybný symbol na vstupu -" +VypisTyp(symbol.typ));
  }
}
```

Vlastnosti metody

Výhody:

- není nutné vytvářet rozkladovou tabulku, třebaže množiny signatur vytvořit musíme,
- nepotřebujeme vlastní zásobník, rekurze probíhá pouze přes vzájemné volání funkcí (procedur) s použitím systémového zásobníku,
- není problém s navázáním sémantické analýzy.

Nevýhody:

- hloubka rekurze může za určitých okolností působit problémy s prostorovou složitostí.

Vlastnosti metody

Výhody:

- není nutné vytvářet rozkladovou tabulku, třebaže množiny signatur vytvořit musíme,
- nepotřebujeme vlastní zásobník, rekurze probíhá pouze přes vzájemné volání funkcí (procedur) s použitím systémového zásobníku,
- není problém s navázáním sémantické analýzy.

Nevýhody:

- hloubka rekurze může za určitých okolností působit problémy s prostorovou složitostí.