10. Translacja sterowana składnią i YACC

10.1 Charakterystyka problemu

- ***translacja sterowana składnią*** jest metodą ***generacji przetworników*** tekstu ***języków***, których ***składnię*** opisano za pomocą ***gramatyki*** (***bezkontekstowej***)

- ***wczytywaniem tekstu wejściowego*** zajmuje się ***parser*** wygenerowany na podstawie gramatyki, natomiast sposób ***generacji tekstu wyjściowego*** zawarty jest w ***akcjach semantycznych*** związanych z produkcjami gramatyki

- z każdym ***symbolem*** gramatyki można związać ***zbiór atrybutów*** (pełniących funkcję zmiennych)

- atrybuty dzielą się na ***syntetyzowane*** („wstępujące” od liści w kierunku korzenia drzewa wywodu) oraz ***dziedziczone*** („zstępujące” od korzenia wzdłuż krawędzi lub od wierzchołka sąsiadującego)

- z każdą ***produkcją*** można związać ***zbiór akcji semantycznych*** – instrukcji (zazwyczaj polegających na obliczaniu wartości atrybutów w celu wygenerowania tekstu wyjściowego)

- opis ***translacji sterowanej składnią*** może przyjąć postać: ***definicji sterowanej składnią***, ***gramatyki atrybutowej*** lub ***schematu translacji***

- ***definicja sterowana składnią*** jest ciągiem ***reguł*** postaci:

  produkcja {*akcje semantyczne*}

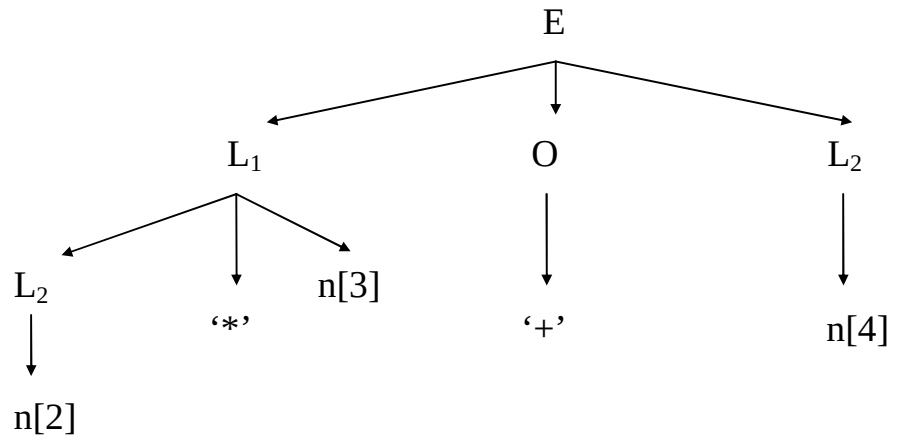
- ***realizacja translacji*** pewnego zdania wejściowego na podstawie definicji ***polega na***:

1) zbudowaniu drzewa wywodu składniowego tego zdania
2) „udekorowaniu” węzłów drzewa atrybutami i akcjami
3) wyznaczeniu (grafu) zależności między atrybutami i porządku wykonywania akcji
4) wykonaniu akcji.

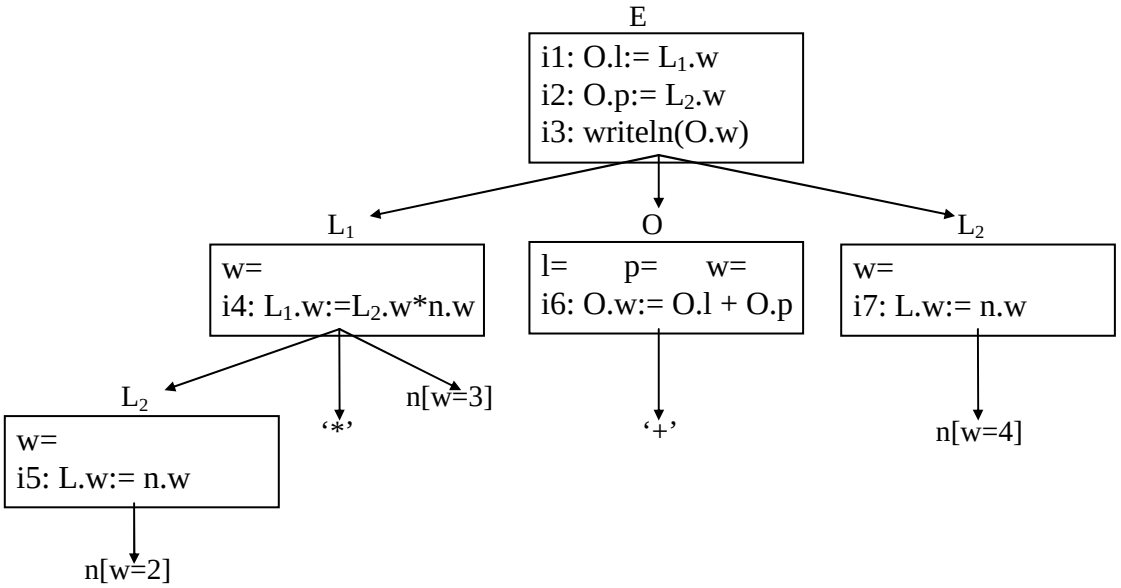
przykład:

Produkcje	*Akcje semantyczne*
$E \rightarrow L_1\ O\ L_2$	O.l := L_1.w; O.p := L_2.w; writeln(O.w)
$O \rightarrow$ ‘+’	O.w := O.l + O.p
$O \rightarrow$ ‘-‘	O.w := O.l - O.p
$L \rightarrow n$	L.w := n.w
$L_1 \rightarrow L_2$ ‘*’ n	L_1.w := L_2.w * n.w

dla ciągu wejściowego: 2*3+4 ***drzewo wywodu składniowego*** przyjmie postać:



natomiast ***drzewo udekorowane***:



akcje semantyczne można zatem wykonać w ***kolejności***

i5, i4, i7, i1, i2, i6, i3

lub

i7, i2, i5, i4, i1, i6, i3

ale nie w kolejności

i1, i2, i3, i4, i5, i6, i7

- definicja sterowana składnią jest ***S-atrybutowa***, jeśli nie ***zawiera*** atrybutów dziedziczonych ***tylko atrybuty syntetyzowane***

- przykład definicji S-atrybutowej:

$E \rightarrow E_1$ ‘+’ T {E.s :=E_1.s + T.s}

$E \rightarrow T$ {E.s :=T.s}

$T \rightarrow n$ {T.s := n.s}

- definicja jest definicją ***L-atrybutową***, jeśli każdy atrybut *p* w każdej akcji semantycznej:

– jest atrybutem syntetyzowanym lub

– jest atrybutem dziedziczonym nieterminala X_i występującego w produkcji $B \rightarrow X_1 X_2 .. X_i .. X_n$ i wartość *p* zależy od wartości atrybutów $X_1 X_2 .. X_{i-1}$ (tzn. od wartości atrybutów symboli występujących na lewo od X_i) lub wartość *p* zależy od atrybutu dziedziczonego symbolu B

- definicje L-atrybutowe mogą być ewaluowane (czyli obliczane) w trakcie *LR-parsingu* bez konieczności budowania drzewa wywodu, ustalania kolejności wykonywania poszczególnych akcji semantycznych itp., co upraszcza i przyspiesza proces translacji

- przykład definicji L-atrybutowej:

Produkcje	*Akcje semantyczne*
D → T L ;	L.t := T.t
T → int	T.t := "int"
T → char	T.t := "char"
L → L_1 ‘,’ id	L_1.t := L.t; writeln(L.t, id.v, ‘;’)
L → id	writeln(L.t, id.v, ‘;’)

- ***definicja sterowana składnią***, w której ***akcje nie generują efektów ubocznych*** (drukowanie, zmienne globalne) nazywa się ***gramatyką atrybutową***
- ***schematem translacji*** nazywamy jej opis, w którym ***należy zaprogramować kolejność wykonywania akcji semantycznych***.

10.2 Translacja w YACCu

- w YACCu programujemy schematy translacji
- ***akcja semantyczna*** ma postać instrukcji języka C i jest ***wykonywana*** w momencie ***redukcji produkcją***, z którą związana jest akcja (drzewo wywodu nie jest tworzone)
- jeśli dana jest produkcja:

$$A: X_1\ X_2\ \ldots\ X_n$$
$$;$$

to atrybut symbolu X_1 jest oznaczany jako `$1`, symbolu X_2 jako `$2` itd., zaś atrybut symbolu stojącego po lewej stronie produkcji (w przykładzie A) jest zawsze oznaczany przez `$$`

- ***przykład schematu translacji*** (plik gram.y):

```
%token n
%%
L : E '='     {printf("%d\n", $1);}
  ;
E : n '+' E   {$$= $1 + $3; }
  | n         {$$= $1;}
  ;
```

analizator leksykalny przyjmuje postać:

```
%{
#include <stdlib.h>
#include "gram.h"
%}
%%
[0-9]+ {yylval=atoi(yytext); return n;}
"+"    {return '+';}
"="    {return '=';}
.|\n   ;
```

- YACC ma ***stos atrybutów*** (v) i stos stanów parsera – obydwa stosy zawierają zawsze tyle samo elementów
- niech t (ang. *top*) oznacza szczyt stosu atrybutów, wtedy operacja *shift* przenosi atrybut (przesuwanego z wejścia na stos stanów) symbolu terminalnego na szczyt stosu v, zaś akcja semantyczna związana z produkcją

$$A: X_1 X_2 \ldots X_{n-1} X_n$$

jest interpretowana w następujący sposób:

1) symbol $n odpowiadający atrybutowi symbolu X_n jest zastępowany przez v[t], $n-1 przez v[t-1], $2 przez v[t-n+2], $1 przez v[t-n+1]

2) symbol $$ jest także zastępowany przez v[t-n+1], co (w trakcie redukcji) odpowiada zdejmowaniu ze stosu parsera n symboli $X_1 \ldots X_n$ i kładzeniu w ich miejsce jednego symbolu A – w takiej sytuacji A znajdzie się na miejscu X_1, więc i atrybut A powinien znaleźć się na miejscu atrybutu X_1

- ***akcje*** poprzedniego schematu translacji są ***zinterpretowane*** następująco:

```
L: E '='    {printf("%d\n", v[t-1]);}
 ;
E: n '+' E {v[t-2] = v[t-2] + v[t]; }
 | n        {v[t] = v[t];}
 ;
```

- jeśli po prawej stronie symbolu produkcji jest tylko jeden symbol, to instrukcja $$=$1 jest równoważna instrukcji pustej (zostanie zawsze przetłumaczona na v[t]=v[t]), zatem można ją pomijać
- ***atrybuty*** mogą przyjmować ***wartości różnych typów***, co zaznaczymy deklaracją postaci (definiuje unię):

```
%union
{ deklaracje typów }
```

- przykład:

```
%union
{char  *tekst;
 int   ival;}
```

- przypisywaniu ***typów atrybutom symboli nieterminalnych*** służy deklaracja:

`%type` <*nazwa_typu*> *nazwa_symbolu*

- ***atrybuty symboli terminalnych*** deklaruje się za pomocą deklaracji `%token`:

`%token` <*nazwa_typu*> *nazwa_terminala*

- **przykład**:

```
%union
{char *text;
}
%type  <text> L
%token <text> id
%token <text> typ
%%
D: L ';'
 ;
L: typ id      {printf("%s %s;\n",$1,$2); $$=$1;}
 | L ',' id    {printf("%s %s;\n",$1,$3); $$=$1;}
 ;
```

analizator leksykalny:

```
%{
#include <stdlib.h>
#include "string.h"
#include "gram.h"
char *p;
%}
%%
"int"|"char"       { yylval.text = strdup(yytext);
                     return typ;
                   }
","                { return ',';}
";"                { return ';';}
[a-zA-Z]+          { p=(char*)calloc(strlen(yytext)+1,sizeof(char));
                     strcpy(p, yytext);
                     yylval.text= p;
                     return id;
                   }
.|\n               ;
```

- załóżmy, że z produkcją

$$B \rightarrow X_1\ X_2\ \ldots\ X_i\ \ldots\ X_n$$

związano akcję semantyczną postaci

$$b = f(p_1, p_2, .., p_i, \ldots\ p_n)$$

1) atrybut b jest atrybutem *syntetyzowanym*, o ile b jest atrybutem B i $p_1, p_2, .., p_n$ są atrybutami symboli $X_1, X_2, .., X_n$

2) jeśli zaś b jest atrybutem X_i i $p_1, p_2, .., p_i, .., p_n$ są atrybutami symboli B, $X_1, X_2, .., X_n$, to b jest atrybutem *dziedziczonym*.

- w YACCu ***jednostki leksykalne*** mają ***tylko atrybuty syntetyzowane***
- **przykład** (z atrybutami dziedziczonymi):

Produkcje	*Akcje semantyczne*	*Atrybuty*
$D \rightarrow T\ L$	L.t:= T.t	L.t – dziedziczony
$T \rightarrow$ int	T.t:= "int"	T.t – syntetyzowany
$T \rightarrow$ char	T.t:= "char"	
$L \rightarrow L_1$, id	L_1.t:= L.t; writeln(L.t, id.v, ';')	
$L \rightarrow$ id	writeln(L.t, id.v, ';')	

- definicja translacji ze zinterpretowanymi na stosie *v* akcjami semantycznymi:

Produkcje	*Akcje semantyczne*
D → T L	
T → int	v[t] := „int”
T→ real	v[t] := „real”
L → id	writeln(v[t-1], v[t])
L → L_1 , id	nt := t-2; writeln(v[t-3], v[t]); t := nt;

- ciąg konfiguracji odpowiadający wejściu: `int a, b, c`

Stos	*Wejście*	*Redukcje*	*Stos v*
$	int id , id , id $		
$ int	id , id , id $	T→ int	v[t] :=”int”
$ T	id , id , id $		
$ T id	, id , id $	L → id	writeln(v[t-1], v[t])
$ T L	, id , id $		
$ T L ,	id , id $		
$ T L , id	, id $	L → L , id	nt := t-2; writeln(v[t-3], v[t]); t := nt;
$ T L	, id $		
$ T L ,	id $		
$ T L , id	$	L → L , id	nt := t-2; writeln(v[t-3], v[t]); t := nt;
$ T L	$	D → T L	

$ D	$		

- ***dziedziczenie atrybutów*** w ***YACCu*** opiera się na ***sięganiu*** w ***głąb stosu atrybutów***; atrybut poniżej atrybutu $1 jest oznaczany w specyfikacjach przez $0, atrybut poprzedzający $0 oznacza się jako $-1 itd.
- w przypadku sięgania w głąb stosu atrybutów konieczne jest bezpośrednie podanie typu atrybutu – atrybut `$0` typu `ival` jest zapisywany jako `$<ival>0;` podobnie można specyfikować w sposób bezpośredni typy atrybutów $$, $1, $2, .., $-1, $-2 itd.

- przykład:

```
%union
{char *text;
};
%type  <text> T
%token <text> id
%token  _int  real
%%
D: T L
 ;
T: _int  {$$ = "int";}
 | real  {$$ = "real";}
 ;
L: id         {printf("%s %s\n", $<text>0,$1);}
 | L ',' id   {printf("%s %s\n", $<text>0,$3);}
 ;
```

- akcje semantyczne można umieszczać pomiędzy symbolami w produkcji (***akcje wielokrotne***):

  X : Y {*akcje_wielokrotne*} Z {*akcje_zwykłe*}
  ;

- powyższa reguła jest zamieniana na:

  X : Y **M** Z {*akcje_zwykłe*}
  ;
  M : {*akcje_wielokrotne*} /* produkcja pusta */
  ;

- każda ***akcja wielokrotna*** jest traktowana jak symbol i ***ma*** przypisany ***numer atrybutu***