Příklad implementace programovacího jazyka

(Mila)

Příklad programu

```
{vypocet nejvetsiho spolecneho delitele}
const a = 84, b = 36;
var x, y;
begin
   x := a; y := b;
   while x <> y do
      if x > y then
       x:= x - y
      else
       y:= y - x;
   write a;
   write b;
   write x;
end
```


EBNF

program = {deklarace} složený-příkaz
deklarace = 'const' identifikátor '=' 'číslo' {',' identifikátor '=' 'číslo'} ';'
| 'var' identifikátor {',' identifikátor} ';'
složený-příkaz = 'begin' příkaz {';' příkaz} 'end'
příkaz = identifikátor-proměnné ':=' výraz
| 'write' výraz
| 'if' podmínka 'then' příkaz ['else' příkaz]
| 'while' podmínka 'do' příkaz
| složený-příkaz
| ε
podmínka = výraz ('=' | '<>' | '<' | '>' | '<=' | '>=') výraz
výraz = ['-'] term {('+' | '-') term}
term = faktor {('*' | '/') faktor}
faktor = identifikátor-proměnné
| identifikátor-konstanty
| číslo
| '(' výraz ')'
identifikátor-proměnné = identifikátor
identifikátor-konstanty = identifikátor

Lexikální elementy

lexikální-element = identifikátor | číslo | spec-symbol
spec-symbol = '+' | '-' | '*' | '/' | '<' | '>'| '<=' | '>=' | ',' | ';' | ':=' | klíčové-slovo
klíčové-slovo = 'var' | 'const' | 'begin' | 'end' | 'if' | 'then' | 'else' | 'while' | 'do' | 'write'
identifikátor = písmeno {písmeno | číslice}
číslo = číslice {číslice}
písmeno = 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
číslice = '0' | '1' | '2' | '3' | '4' | '5' | '6

Zásobníkový počítač

zásobník	z : array [0..maxz] of integer
vrch zásobníku	v : integer
paměť proměnných	m : array [0..maxm] of integer
čítač instrukcí	ci : integer

typ instr.	operand	význam instrukce	provedení instrukce
TC	*n*	uložení hodnoty na zásobník	v := v + 1; z[v] := n
TA	*a*	uložení adresy na zásobník	v := v + 1; z[v] := a
DR		dereference	z[v] := m[z[v]]
BOP	*op*	binární operace	v := v-1; z[v] := z[v] *op* z[v+1]
UNM		unární minus	z[v] := -z[v]
ST		přiřazení	m[z[v-1]] := z[v]; v := v – 2
WRT		výstup hodnoty	write(z[v]); v := v – 1
JU	*adr*	nepodmíněný skok	ci := *adr*
IFJ	*adr*	podmíněný skok	b := z[v]; v := v-1; if b=0 then ci := *adr*
STOP		ukončení programu	

operátor	symb. označení	číselný kód
+	*Plus*	0
-	*Minus*	1
*	*Times*	2
/	*Divide*	3
=	*Eq*	4
<>	*NotEq*	5
<	*Less*	6
>	*Greater*	7
<=	*LessOrEq*	8
>=	*GreaterOrEq*	9

Příklad

```
          const a = 84, b = 36;
          var x, y;   {adresa x je 0, adresa y je 1}
          begin
 0: TA  0      x := a;
 1: TC  84
 2: ST
 3: TA  1      y := b;
 4: TC  36
 5: ST
 6: TA  0      while x <> y do
 7: DR
 8: TA  1
 9: DR
10: BOP 5
11: IFJ 34
12: TA  0        if x > y then
13: DR
14: TA  1
15: DR
16: BOP 7
17: IFJ 26
18: TA  0           x := x - y
```



```
19: TA  0
20: DR
21: TA  1
22: DR
23: BOP 1
24: ST
25: JU  33         else
26: TA  1           y := y - x;
27: TA  1
28: DR
29: TA  0
30: DR
31: BOP 1
32: ST
33: JU  6
34: TC  84       write a;
35: WRT
36: TC  36       write b;
37: WRT
38: TA  0      write x;
39: DR
40: WRT
41: STOP     end
```


Interpret

Specifikace:

```
enum TypInstr {TA, TC, BOP, UNM, DR, ST, IFJ, JU, WRT, STOP
};
enum Operator {
   Plus, Minus, Times, Divide,
   Eq, NotEq, Less, Greater, LessOrEq, GreaterOrEq
};
void Run();
```

Implementace:

```
struct Instr {
   TypInstr typ;
   int      opd;
};
enum {MaxZas = 100, MaxProm = 100, MaxProg = 200};

static int z[MaxZas];          // zasobnik
static int v;                  // vrchol zasobniku
static int m[MaxProm];         // pamet promennych
static Instr p[MaxProg];       // pamet programu;
static int ic;                 // citac instrukci
```

Interpret

```
void Run()
{
   Instr instr;
   printf("\nInterpretace programu\n");
   ic = 0; v = -1;
   for (;;) {
      instr = p[ic++];
      switch (instr.typ) {
      case TA:
         z[++v] = instr.opd; break;
      case TC:
         z[++v] = instr.opd; break;
      case BOP: {
         int right = z[v--];
         int left = z[v--];
         switch (instr.opd) {
         case Plus:
            z[++v] = left + right; break;
         case Minus:
            ...
         case Eq:
            z[++v] = left == right; break;
         case NotEq:
```



```
            ...
         }
         break;
      }
      case UNM:
         z[v] = -z[v]; break;
      case DR:
         z[v] = m[z[v]]; break;
      case ST: {
         int val = z[v--];
         int adr = z[v--];
         m[adr] = val; break;
         }
      case IFJ:
         if (!z[v--]) ic = instr.opd; break;
      case JU:
         ic = instr.opd; break;
      case WRT:
         printf("%d\n", z[v--]); break;
      case STOP:
         printf("Konec interpretace\n\n"); return;
      }
   }
}
```


Lexikální analyzátor

```
typedef enum {
/* ident  číslo +     -      *      /       */
   IDENT, NUMB, PLUS, MINUS, TIMES, DIVIDE,
/* =   <>   <   >   <=   >=   (     )     :=    */
   EQ, NEQ, LT, GT, LTE, GTE, LPAR, RPAR, ASSGN,
/* ,      ;                    */
   COMMA, SEMICOLON,
/* var    const    begin    end    if    then    else */
   kwVAR, kwCONST, kwBEGIN, kwEND, kwIF, kwTHEN, kwELSE,
/* while    do    write                  */
   kwWHILE, kwDO, kwWRITE,
/* konec souboru                      */
   EOI
} LexSymbol;

enum {MaxLenIdent = 32};
extern LexSymbol Symb;
extern char      Ident[MaxLenIdent];  /* atribut symbolu IDENT */
extern int       Cislo;               /* atribut symbolu NUMB */

void CtiSymb(void);
void InitLexan(char*);
void Chyba(char*);
void ChybaSrovnani(LexSymbol);
```


Převod EBNF na bezkontextovou gramatiku

pravidla EBNF	pravidla bezkontextové gramatiky
$A = \alpha\,(\beta_1 \mid \beta_2)\,\chi$	$A \to \alpha X \chi$ $X \to \beta_1$ $X \to \beta_2$ nebo $A \to \alpha \beta_1 \chi$ $A \to \alpha \beta_2 \chi$
$A = \alpha\,[\beta]\,\chi$	$A \to \alpha X \chi$ $X \to \beta$ $X \to \varepsilon$ nebo $A \to \alpha \beta \chi$ $A \to \alpha \chi$
$A = \alpha\,\{\beta\}\,\chi$	$A \to \alpha X \chi$ $X \to \beta X$ $X \to \varepsilon$

LL*(1)* gramatika

1: Program → Dekl SložPříkaz
2: Dekl → DeklKonst Dekl
3: Dekl → DeklProm Dekl
4: Dekl → ε
5: DeklKonst → const ident = číslo ZbDeklKonst ;
6: ZbDeklKonst → , ident = číslo ZbDeklKonst
7: ZbDeklKonst → ε
8: DeklProm → var ident ZbDeklProm ;
9: ZbDeklProm → , ident ZbDeklProm
10: ZbDeklProm → ε
11: SložPříkaz → begin Příkaz ZbPříkazů end
12: ZbPříkazů → ; Příkaz ZbPříkazů
13: ZbPříkazů → ε
14: Příkaz → ident := Výraz
15: Příkaz → write Výraz
16: Příkaz → if Podmínka then Příkaz ČástElse
17: Příkaz → while Podmínka do Příkaz

18: Příkaz → SložPříkaz
19: Příkaz → ε
20: ČástElse → else Příkaz
21: ČástElse → ε
22: Podmínka → Výraz RelOp Výraz
23: RelOp → =
24: RelOp → <>
25: RelOp → <
26: RelOp → >
27: RelOp → <=
28: RelOp → >=
29: Výraz → Term ZbVýrazu
30: Výraz → - Term ZbVýrazu
31: ZbVýrazu → + Term ZbVýrazu
32: ZbVýrazu → - Term ZbVýrazu
33: ZbVýrazu → ε
34: Term → Faktor ZbTermu
35: ZbTermu → * Faktor ZbTermu
36: ZbTermu → / Faktor ZbTermu

37: ZbTermu → ε
38: Faktor → ident
39: Faktor → číslo
40: Faktor → (Výraz)

Překlad do jazyka zásobníkového počítače

Atributy vstupních symbolů:

vstupní symbol	syntetizovaný atribut
ident	sid
číslo	shod

Atributy výstupních symbolů:

výstupní symbol	dědičný atribut
TC	dhod
TA	da
DR	
BOP	dop
UNM	
ST	
WRT	
JU	dadr
IFJ	dadr
STOP	

Zpracování deklarací a tabulka symbolů

Tabulka symbolů:

klíč: identifikátor

hodnota: popis významu identifikátoru

Jména procedur operujících nad tabulkou symbolů budeme považovat za speciální výstupní symboly, jejichž výskyt v pravidle překladové gramatiky znamená vyvolání příslušné procedury.

Dědičné atributy těchto symbolů budou reprezentovat vstupní parametry procedur.

výst. symbol	dědičné atr.	význam
deklKonst	did, dhod	Symbol označuje proceduru *deklKonst*(*id*,*h*), která doplní do tabulky symbolů deklaraci identifikátoru konstanty *id* s hodnotou *h*. Jde-li o druhou deklaraci identifikátoru, procedura ohlásí chybu a tabulku symbolů nezmění.
deklProm	did	Symbol označuje proceduru *deklProm*(*id*), která doplní do tabulky symbolů deklaraci identifikátoru proměnné *id*. Deklarovaným proměnným jsou postupně přidělovány adresy 0, 1, 2, ... Jde-li o druhou deklaraci identifikátoru, procedura ohlásí chybu a tabulku symbolů nezmění.

1:	Program → Dekl SložPříkaz STOP	
2:	Dekl → DeklKonst Dekl	
3:	Dekl → DeklProm Dekl	
4:	Dekl → ε	
5:	DeklKonst → const ident = číslo *deklKonst* ZbDeklKonst ;	*deklKonst*.did := ident.sid *deklKonst.dhod* := číslo.shod
6:	ZbDeklKonst → , ident = číslo *deklKonst* ZbDeklKonst	*deklKonst*.did := ident.sid *deklKonst.dhod* := číslo.shod
7:	ZbDeklKonst → ε	
8:	DeklProm → var ident *deklProm* ZbDeklProm ;	*deklProm*.did := ident.sid
9:	ZbDeklProm → , ident *deklProm* ZbDeklProm	*deklProm*.did := ident.sid
10:	ZbDeklProm → ε	

Překlad výrazů

Snadný, až na pravidlo

Faktor → ident

kde překlad závisí na tom, jak je identifikátor deklarován.

Jestliže *ident* je identifikátorem proměnné, výstupem má být dvojice instrukcí

TA *a*

DR

kde *a* je adresa proměnné.

Jestliže *ident* je identifikátorem konstanty, výstupem má být instrukce

TC *hod*

kde *hod* je hodnota konstanty.

Toto rozlišení nelze provést pomocí pravidel překladové gramatiky.

Zavedeme proto speciální výstupní symbol ***TR*** (take result) označující sémantickou proceduru, která z tabulky symbolů zjistí, jak je identifikátor deklarován, a vygeneruje odpovídající instrukce. Vstupním parametrem procedury bude identifikátor, symbol ***TR*** tedy bude mít jediný dědičný atribut *did*.

výst. symbol	dědičný atr.	význam
TR	did	Symbol označuje proceduru *GenTR*(*id*), která zjistí, zda identifikátor *id* je deklarován jako identifikátor proměnné nebo konstanty. V prvním případě generuje instrukce TA *a* DR kde *a* je adresa proměnné. Ve druhém případě generuje instrukci TC *h* kde *h* je hodnota konstanty. Není-li identifikátor deklarován, procedura ohlásí chybu.

22:	Podmínka → Výraz RelOp Výraz BOP	BOP.dop := RelOp.sop
23:	RelOp → =	RelOp.sop := *Eq*
24:	RelOp → <>	RelOp.sop := *NotEq*
...	...	...
28:	RelOp → >=	RelOp.sop := *GreaterOrEq*
29:	Výraz → Term ZbVýrazu	
30:	Výraz → - Term UNM ZbVýrazu	
31:	ZbVýrazu → + Term BOP ZbVýrazu	BOP.dop := *Plus*
32:	ZbVýrazu → - Term BOP ZbVýrazu	BOP.dop := *Minus*
33:	ZbVýrazu → ε	
34:	Term → Faktor ZbTermu	
35:	ZbTermu → * Faktor BOP ZbTermu	BOP.dop := *Times*
36:	ZbTermu → / Faktor BOP ZbTermu	BOP.dop := *Divide*
37:	ZbTermu → ε	
38:	Faktor → ident *TR*	*TR*.did := ident.sid
39:	Faktor → číslo TC	TC.dhod := číslo.shod
40:	Faktor → (Výraz)	

Překlad jednoduchých příkazů

Přiřazovací příkaz

ident := *výraz*

se přeloží do posloupnosti instrukcí

TA *a*
instrukce vytvořené překladem výrazu
ST

kde *a* je adresa proměnné *ident*.

Pro zjištění adresy proměnné zavedeme funkci *adrProm* nad tabulkou symbolů:

volání funkce	význam
adrProm(*id*)	Funkce zjistí, zda *id* je identifikátorem proměnné. V kladném případě vrátí adresu proměnné, v záporném případě ohlásí chybu a vrátí nulu.

14:	Příkaz → ident TA := Výraz ST	TA.da := *adrProm*(ident.sid)

Příkaz výstupu

write *výraz*

se přeloží do posloupnosti instrukcí

instrukce vytvořené překladem výrazu
WRT

15:	Příkaz → write Výraz WRT	

Prázdný příkaz má prázdný překlad.

19:	Příkaz → ε	

Překlad příkazu while

Příkaz cyklu

while *podmínka* do *příkaz*

se přeloží do posloupnosti instrukcí

*a*1: *instrukce vytvořené překladem podmínky*
IFJ *a*2
instrukce vytvořené překladem příkazu
JU *a*1
*a*2:

Překlad vytvoříme tímto způsobem:

- před zpracováním podmínky si zapamatujeme aktuální hodnotu čítače instrukcí generovaného programu (tj. adresu *a*1),
- po překladu podmínky vygenerujeme instrukci *IFJ* s prozatím neurčeným operandem a zapamatujeme si adresu instrukce v cílovém programu (označme ji *aIFJ*),
- po překladu příkazu tvořícího tělo cyklu vygenerujeme instrukci *JU* s operandem *a*1,
- do operandu instrukce na adrese *aIFJ* dosadíme aktuální hodnotu čítače instrukcí generovaného programu (*a*2).

Překlad příkazu while

Zavedeme speciální výstupní symboly *GetIC* a *PutIC* označující pomocné sémantické akce operující s čítačem instrukcí generovaného programu.

výst. symbol	děd. atr.	synt. atr.	význam
GetIC		sadr	Symbol označuje funkci *GetIC*(), která vrátí aktuální hodnotu čítače instrukcí generovaného programu
PutIC	dadr		Symbol označuje proceduru *PutIC*(*adr*), která dosadí do operandu instrukce na adrese *adr* aktuální hodnotu čítače instrukcí generovaného programu

Adresu vygenerované instrukce *IFJ* formálně označíme tak, že k tomuto symbolu přidružíme syntetizovaný atribut *sadr*.

17:	Příkaz → while *GetIC* Podmínka IFJ do Příkaz JU *PutIC*	IFJ.dadr := 0 JU.dadr := *GetIC*.sadr *PutIC*.dadr := IFJ.sadr

Překlad příkazu if

příkaz	překlad
if *podmínka* then *příkaz*1	*instrukce vytvořené překladem podmínky* IFJ *a*1 *instrukce vytvořené překladem příkazu*1 *a*1:
if *podmínka* then *příkaz*1 else *příkaz*2	*instrukce vytvořené překladem podmínky* IFJ *a*1 *instrukce vytvořené překladem příkazu*1 JU *a*2 *a*1: *instrukce vytvořené překladem příkazu*2 *a*2:

Využijeme speciální výstupní symboly *GetIC* a *PutIC*.

Úpravu instrukce *IFJ* provedeme až v pravidle pro neterminál *ČástElse*. K tomuto neterminálu proto přidružíme dědičný atribut *dadr*, jehož hodnotou bude adresa instrukce *IFJ*, která má být upravena.

Překlad příkazu if

16:	Příkaz → if Podmínka IFJ then Příkaz ČástElse	IFJ.dadr = 0 ČástElse.dadr := IFJ.sadr
20:	ČástElse → else JU $PutIC^1$ Příkaz $PutIC^2$	JU.dadr := 0 $PutIC^1$.dadr := ČástElse.dadr $PutIC^2$.dadr := JU.sadr
21:	ČástElse → *PutIC*	*PutIC*.dadr := ČástElse.dadr

Přehled přidružení atributů

druh symbolu	symbol	dědičné atributy	syntetiz. atributy
neterminální symbol	Program, Dekl, DeklKonst, DeklProm, ZbDeklKonst, ZbDeklProm, SložPříkaz, ZbPříkazů, Příkaz, Podmínka, Výraz, ZbVýrazu, Term, ZbTermu, Faktor		
	ČástElse	dadr	
	RelOp		sop
vstupní symbol	ident		sid
	číslo		shod
výstupní symbol označující instrukci zásobníkového počítače	TC	dhod	
	TA	da	
	DR		
	BOP	dop	
	UNM		
	ST		
	WRT		
	JU	dadr	sadr
	IFJ	dadr	sadr
	STOP		

speciální výstupní symbol	*deklKonst*	did, dhod	
	deklProm	did	
	TR	did	
	GetIC		sadr
	PutIC	dadr	

Realizace tabulky symbolů

Spojový seznam, jehož prvky budou (kromě ukazatele na další prvek) obsahovat:

- lexikální element identifikátoru (řetěz znaků)
- kód druhu identifikátoru
- celé číslo označující adresu proměnné nebo hodnotu konstanty.

```
enum DruhId {Nedef, IdProm, IdKonst};
struct PrvekTab {
   char *ident;
   DruhId druh;
   int hodn;
   PrvekTab *dalsi;
   PrvekTab(char *i, DruhId d, int h, PrvekTab *n);
};

PrvekTab::PrvekTab(char *i, DruhId d, int h, PrvekTab *n)
{
   ident = new char[strlen(i)+1];
   strcpy(ident, i);
   druh = d; hodn = h; dalsi = n;
};
```


Realizace tabulky symbolů

```
static PrvekTab *TabSym; /* ukazatel na začátek tab. symbolů */

static PrvekTab *hledejId(char *id)
{
   PrvekTab *p = TabSym;
   while (p)
      if (strcmp(id,p->ident)==0)
         return p;
      else
         p = p->dalsi;
   return NULL;
}

static void Chyba(char *id, char *txt)
{
   printf("identifikator %s: %s\n", id, txt);
}
```


Realizace tabulky symbolů

```
void deklKonst(char *id, int val)
{
   PrvekTab *p = hledejId(id);
   if (p) {
      Chyba(id, "druha deklarace");
      return;
   }
   TabSym = new PrvekTab(id, IdKonst, val, TabSym);
}

void deklProm(char *id)
{
   static int volna_adr;
   PrvekTab *p = hledejId(id);
   if (p) {
      Chyba(id, "druha deklarace");
      return;
   }
   TabSym = new PrvekTab(id, IdProm, volna_adr, TabSym);
   volna_adr++;
}
```

Realizace tabulky symbolů

```
int adrProm(char *id)
{
   PrvekTab *p = hledejId(id);
   if (!p) {
      Chyba(id, "neni deklarovan");
      return 0;
   } else if (p->druh != IdProm) {
      Chyba(id, "neni identifikatorem promenne");
      return 0;
   } else
      return p->hodn;
}

DruhId idPromKonst(char *id, int *v)
{
   PrvekTab *p = hledejId(id);
   if (p) {
      *v = p->hodn;
      return p->druh;
   }
   Chyba(id, "neni deklarovan");
   return Nedef;
}
```


Realizace výstupních funkcí

Specifikace:

```
int Gener(TypInstr, int = 0);
void GenTR(char*);
void PutIC(int);
int GetIC();
```

Implementace:

```
int Gener(TypInstr ti, int opd)
{
   p[ic].typ = ti;
   p[ic].opd = opd;
   return ic++;
}
```


Realizace výstupních funkcí

```
void GenTR(char *id)
{
   int v;
   DruhId druh = idPromKonst(id, &v);
   switch (druh) {
   case IdProm:
      Gener(TA, v);
      Gener(DR);
      break;
   case IdKonst:
      Gener(TC, v);
      break;
   }
}

void PutIC(int adr)
{  p[adr].opd = ic; }

int GetIC()
{  return ic; }
```


Rekurzívní sestup

```
void Srovnani(LexSymbol s)
{
   if (Symb == s)
      CtiSymb();
   else
      ChybaSrovnani(s);
}

void Srovnani_IDENT(char *id)
{
   if (Symb == IDENT) {
      strcpy(id, Ident); CtiSymb();
   } else
      ChybaSrovnani(IDENT);
}
void Srovnani_NUMB(int *h)
{
   if (Symb == NUMB) {
      *h = Cislo; CtiSymb();
   } else
      ChybaSrovnani(NUMB);
}
```

Rekurzívní sestup

```
void Program()
{  /* Program -> Dekl SložPříkaz STOP */
   Dekl(); SlozPrikaz(); Gener(STOP);
}

void Dekl()
{
   switch (Symb) {
   case kwVAR:    /* Dekl -> DeklProm Dekl */
      DeklProm(); Dekl();
      break;
   case kwCONST:  /* Dekl -> DeklKonst Dekl */
      DeklKonst(); Dekl();
      break;
   default:       /* Dekl -> ε */
      ;
   }
}
```


Rekurzívní sestup

```
void DeklKonst()
{  /* DeklKonst -> const ident = číslo deklKonst ZbDeklKonst ; */
   char id[MaxLenIdent];
   int hod;
   CtiSymb();
   Srovnani_IDENT(id); Srovnani(EQ); Srovnani_NUMB(&hod);
   deklKonst(id, hod); ZbDeklKonst(); Srovnani(SEMICOLON);
}

void ZbDeklKonst()
{
   if (Symb == COMMA) {
   /* ZbDeklKonst -> , ident = číslo deklKonst ZbDeklKonst */
      char id[MaxLenIdent];
      int hod;
      CtiSymb();
      Srovnani_IDENT(id); Srovnani(EQ); Srovnani_NUMB(&hod);
      deklKonst(id, hod); ZbDeklKonst();
   }
   else
   /* ZbDeklKonst -> ε */ ;
}
```


Rekurzívní sestup

```
void DeklProm()
{  /* DeklProm -> var ident deklProm ZbDeklProm ; */
   char id[MaxLenIdent];
   CtiSymb();
   Srovnani_IDENT(id); deklProm(id); ZbDeklProm();
   Srovnani(SEMICOLON);
}

void ZbDeklProm()
{
   if (Symb == COMMA) {
   /* ZbDeklProm -> , ident deklProm ZbDeklProm */
      char id[MaxLenIdent];
      CtiSymb();
      Srovnani_IDENT(id); deklProm(id); ZbDeklProm();
   }
   else
   /* ZbDeklProm -> ε */ ;
}
```


Rekurzívní sestup

```
void SlozPrikaz()
{  /* begin Příkaz ZbPříkazů end */
   Srovnani(kwBEGIN); Prikaz(); ZbPrikazu(); Srovnani(kwEND);
}

void ZbPrikazu()
{
   if (Symb == SEMICOLON) {
   /* ZbPrikazu -> ; Příkaz ZbPříkazů */
      CtiSymb(); Prikaz(); ZbPrikazu();
   }
   else
   /* ZbPrikazu -> ε */ ;
}
```

Rekurzívní sestup

```
void Prikaz()
{
   switch (Symb) {
   case IDENT: {
   /* Příkaz -> ident TA := Výraz ST */
      Gener(TA, adrProm(Ident)); CtiSymb();
      Srovnani(ASSGN); Vyraz(); Gener(ST);
      break;
   }
   case kwWRITE:
   /* Příkaz -> write Výraz WRT */
      CtiSymb(); Vyraz(); Gener(WRT);
      break;
   case kwIF: {
   /* Příkaz -> if Podmínka IFJ then Příkaz ČástElse */
      CtiSymb(); Podminka();
      int adrIFJ = Gener(IFJ);
      Srovnani(kwTHEN); Prikaz(); CastElse(adrIFJ);
      break;
   }
```


Rekurzívní sestup

```
   case kwWHILE: {
   /* Příkaz -> while GetIC Podmínka IFJ do Příkaz JU PutIC */
      int a1 = GetIC();
      CtiSymb(); Podminka();
      int aIFJ = Gener(IFJ);
      Srovnani(kwDO); Prikaz(); Gener(JU, a1); PutIC(aIFJ);
      break;
   }
   case kwBEGIN:
   /* Příkaz -> SložPříkaz */
      SlozPrikaz();
      break;
   }
}
```


Rekurzívní sestup

```
void CastElse(int adrIFJ)
{
   if (Symb == kwELSE) {
   /* ČástElse -> else JU PutIC Příkaz PutIC */
      CtiSymb();
      int adrJU = Gener(JU);
      PutIC(adrIFJ); Prikaz(); PutIC(adrJU);
   }
   else
   /* ČástElse -> ε */ ;
}

void Podminka()
{  /* Podmínka -> Výraz RelOp Výraz BOP */
   Vyraz();
   Operator op = RelOp();
   Vyraz();
   Gener(BOP, op);
}
```


Rekurzívní sestup

```
Operator RelOp()
{
   switch (Symb) {
   case EQ:    /* RelOp -> = */
      CtiSymb(); return Eq;
   case NEQ:   /* RelOp -> <> */
      CtiSymb(); return NotEq;
   case LT:    /* RelOp -> < */
      CtiSymb(); return Less;
   case GT:    /* RelOp -> > */
      CtiSymb(); return Greater;
   case LTE:   /* RelOp -> <= */
      CtiSymb(); return LessOrEq;
   case GTE:   /* RelOp -> >= */
      CtiSymb(); return GreaterOrEq;
   default:
      Chyba("neocekavany symbol");
   }
}
```

Rekurzívní sestup

```
void Vyraz()
{
   if (Symb == MINUS) {
   /* Výraz -> - Term UNM ZbVýrazu */
      CtiSymb(); Term(); Gener(UNM); ZbVyrazu();
   } else {
   /* Výraz -> Term ZbVýrazu */
      Term(); ZbVyrazu();
   }
}

void ZbVyrazu()
{
   switch (Symb) {
   case PLUS:   /* ZbVýrazu -> + Term BOP ZbVýrazu */
      CtiSymb(); Term(); Gener(BOP, Plus); ZbVyrazu(); break;
   case MINUS:  /* ZbVýrazu -> - Term BOP ZbVýrazu */
      CtiSymb(); Term();  Gener(BOP, Minus); ZbVyrazu(); break;
   case default: /* ZbVýrazu -> ε */ ;
   }
}
```


Rekurzívní sestup

```
void Term()
{  /* Term -> Faktor ZbTermu */
   Faktor(); ZbTermu();
}

void ZbTermu()
{
   switch (Symb) {
   case TIMES: /* ZbTermu -> * Faktor BOP ZbTermu */
      CtiSymb(); Faktor(); Gener(BOP, Times); ZbTermu();
      break;
   case DIVIDE: /* ZbTermu -> / Faktor BOP ZbTermu */
      CtiSymb(); Faktor(); Gener(BOP, Divide); ZbTermu();
      break;
   case default:/* ZbTermu -> ε */ ;
   }
}
```


Rekurzívní sestup

```
void Faktor()
{
   switch (Symb) {
   case IDENT: /* Faktor -> ident TR */
      char id[MaxLenIdent];
      Srovnani_IDENT(id); GenTR(id);
      break;
   case NUMB:  /* Faktor -> číslo TC */
      int hodn;
      Srovnani_NUMB(&hodn); Gener(TC, hodn);
      break;
   case LPAR:  /* Faktor -> ( Výraz ) */
      CtiSymb(); Vyraz(); Srovnani(RPAR);
      break;
   default:
      Chyba("Neocekavany symbol");
   }
}
```