YACC

Postfixová kalkulačka

postfix.l:

```
%{
#include < stdlib.h >
#include < stdio.h >
#include "y.tab.h"

#define YYSTYPE int
extern YYSTYPE yylval;
%}
%%
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
[ \t]+  ;
\n      return ENTER;
.       return (int) yytext[0];
%%
```

postfix.y:

```
%{
#include < stdio.h >
#include < stdlib.h >
#define YYSTYPE int
%}
%token NUMBER ENTER
%%
session:        /* empty */
                | session line
                ;
line:           expression ENTER { printf("---- %d\n", $1); }
                ;
expression:     NUMBER { $$ = $1; }
                | expression expression '+' { $$ = $1 + $2; }
                | expression expression '-' { $$ = $1 - $2; }
                | expression expression '*' { $$ = $1 * $2; }
                | expression expression '/' { $$ = $1 / $2; }
                ;
%%
int main(int argc, char **argv)
{
  yyparse();
  return 0;
}

int yyerror(char *s)
{
  printf("error: %s\n", s);
}

int yywrap(void)
{
  return 1;
}
```

překlad:

```
yacc -d postfix.y
flex postfix.l
gcc -o postfix y.tab.c lex.yy.c
```

Infixová kalkulačka

infix.y:

```
%{
 #include < stdio.h >
 #include < stdlib.h >
 #define YYSTYPE int
%}
%token NUMBER ENTER
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
session:        /* empty */
                | session line
                ;
line:           expression ENTER { printf("---- %d\n", $1); }
                ;
expression:     NUMBER { $$ = $1; }
                | expression '+' expression   { $$ = $1 + $3; }
                | expression '-' expression   { $$ = $1 - $3; }
                | expression '*' expression   { $$ = $1 * $3; }
                | expression '/' expression   { $$ = $1 / $3; }
                | '(' expression ')'          { $$ = $2; }
                | '-' expression %prec UMINUS { $$ = -$2; }
                ;
%%
int main(int argc, char **argv)
```

```
{
  yyparse();
  return 0;
}

int yyerror(char *s)
{
  printf("error: %s\n", s);
}

int yywrap(void)
{
  return 1;
}
```

Úlohy k procvičení

1. **Předělejte stávající kalkulačku tak, aby nevyhodnocovala výrazy, ale aby místo toho generovala instrukce PL/0. Například pro výraz 1+2*3 by se měly generovat instrukce**

   ```
   LIT 0, 1
   LIT 0, 2
   LIT 0, 3
   OPR 0, 4
   OPR 0, 2
   ```

2. **Doplňte generovaný kód tak, aby byl kompatibilní s PL/0 interpretem či debuggerem. Například pro výraz "(1 + 2) * 3" se vygenerují instrukce**

   ```
   0  INT 0, 3
   1  LIT 0, 1
   2  LIT 0, 2
   3  OPR 0, 2
   4  LIT 0, 3
   5  OPR 0, 4
   6  RET 0, 0
   ```

3. **Předpokládejme, že máme definované proměnné r0 až r9. Upravte lexikální analyzátor tak, aby dokázal takové identifikátory načítat. Pak upravte syntaktický analyzátor tak, aby dokázal zpracovávat přiřazovací příkazy typu** `identifikátor := výraz;`**. Například pro příkaz "r0 := (1 + 2) * r1;" se vygenerují instrukce**

   ```
   0  INT 0, 13
   1  LIT 0, 1
   2  LIT 0, 2
   3  OPR 0, 2
   4  LOD 0, 4
   5  OPR 0, 4
   6  STO 0, 3
   7  RET 0, 0
   ```

4. **Doplňte do kalkulačky, která se stává programovacím jazykem, podmíněný příkaz** `if-then`**. Kvůli větvi** `if` **budete muset rozšířit lexikální analyzátor o tokeny** `<`**,** `>`**,** `=`**,** `<=`**,** `>=` **a** `<>`**. Například program**

   ```
   r0 := 2;
   r1 := 4;
   if r0-r1 < r0+r1 then r2 := 1;
   ```

   se přeloží do instrukcí

   ```
    0  INT 0, 13
    1  LIT 0, 2
    2  STO 0, 3
    3  LIT 0, 4
    4  STO 0, 4
    5  LOD 0, 3
    6  LOD 0, 4
    7  OPR 0, 3
    8  LOD 0, 3
    9  LOD 0, 4
   10  OPR 0, 3
   11  OPR 0, 10
   12  JMC 0, 15
   13  LIT 0, 1
   14  STO 0, 5
   15  RET 0, 0
   ```

Překladač PL/0

Pokud jste se dostali až sem a chcete vědět, zda můžete yacc/bison použít pro implementaci skutečného překladače (například PL/0), pokusíme se odpovědět příkladem. Nejprve se pokusíme o velmi jednoduchou podmnožinu jazyka PL/0, překladač pak budeme rozšiřovat a uvidíme, kam se dostaneme.

Nejprve se pokusíme o jednoduchý jazyk, který umí jen definovat proměnné příkazem `var` a v bloku `begin-end` (který může být i vnořený v jiném) do nich přiřazovat výsledky aritmetických operací. Začněme lexikálním analyzátorem:

pl0_prirad.l

```
%{
#include < stdlib.h >
#include < stdio.h >
```

```
#include "pl0_prirad.tab.h"

%}
%option bison-bridge bison-locations
%%
[0-9]+               { yylval->cislo = atoi(yytext); return CISLO; }
begin                { return K_BEGIN; }
end                  { return K_END; }
var                  { return K_VAR; }
:=                   { return PRIRAD; }
\+                   { return O_PLUS; }
\-                   { return O_MINUS; }
\*                   { return O_KRAT; }
\/                   { return O_DELENO; }
[a-zA-Z][a-zA-Z0-9]* { yylval->retezec = strdup(yytext); return IDENTIFIKATOR; }
\.                   { return TECKA; }
,                    { return CARKA; }
;                    { return STREDNIK; }
\(                   { return ZAVORKA_L; }
\)                   { return ZAVORKA_P; }
[ \t\12\15]+         ;
.                    { return (int) yytext[0]; }
%%
```

Jediná neznámé konstrukce jsou `%option bison-bridge bison-locations`, která souvisí s typem sémanticé hodnoty rozpoznaného tokenu; rozpoznáme-li číslo, chceme vrátit jeho hodnotu, rozpoznáme-li identifikátor, chceme vrátit jeho název.

Syntaktický analyzátor bude potřebovat více komentáře:

pl0_prirad.y

```
%{
#include < stdio.h >
#include < stdlib.h >
#include < string.h >
```

Budeme potřebovat různé pomocné funkce, definujme-si jejich návratové hodnoty:

```
#define OK      0
#define CHYBA   -1
```

Generované instrukce si budeme pamatovat v paměti pro kód; z překladače PL/0 totiž víme, že v okamžiku generování instrukce nemusí být ještě známy její parametry.

```
#define MAX_DELKA_KODU  1000
int kod[MAX_DELKA_KODU][3];
int delka_kodu = 0;
```

Pro usnadnění práce si nadefinujeme symbolické pojmenování instrukcí, pro výpis pak jejich vyjádření řetězcem.

```
#define LIT   0
#define OPR   1
#define LOD   2
#define STO   3
#define CAL   4
#define RET   5
#define INT   6
#define JMP   7
#define JMC   8

#define NEG   1
#define ADD   2
#define SUB   3
#define MUL   4
#define DIV   5
#define MOD   6
#define ODD   7
#define EQ    8
#define NE    9
#define LT   10
#define GE   11
#define GT   12
#define LE   13

#define POCET_INSTRUKCI         9
#define MAX_PISMEN_INSTRUKCE    3
typedef char NAZEV_INSTRUKCE[MAX_PISMEN_INSTRUKCE+1];
typedef int INSTRUKCE[3];

NAZEV_INSTRUKCE nazev_instrukce[POCET_INSTRUKCI] =
{"LIT", "OPR", "LOD", "STO", "CAL", "RET", "INT", "JMP", "JMC"};
```

Také si napíšeme jednoduché funkce pro vložení (generování) instrukce do paměti kódu a pro výpis celkého kódu na obrazovku:

```
void gen(int instr, int param1, int param2)
{
        kod[delka_kodu][0] = instr;
        kod[delka_kodu][1] = param1;
        kod[delka_kodu][2] = param2;
        delka_kodu++;
}


void vypis_kod(void)
{
```

```
        int i;
        for (i=0; i < delka_kodu; i++)
                printf("%3d %s %d %d\n",
                                i,
                                nazev_instrukce[kod[i][0]],
                                kod[i][1],
                                kod[i][2]);
}
```

V tomto příkladu jsou jediné typy identifikátorů názvy proměnných, časem ale budeme uvažovat i jiné typy. Už teď tedy zavedeme tabulku obecných identifikátorů. Zatím nebudeme rozlišovat typ ani jejich umístění v paměti; to doplníme časem. Jediné, co tedy budeme evidovat, bude název identifikátoru.

```
#define MAX_POCET_IDENTIFIKATORU        100
#define MAX_DELKA_IDENTIFIKATORU        30
typedef struct {
 char nazev[MAX_DELKA_IDENTIFIKATORU+1];
 } T_IDENTIFIKATOR;

T_IDENTIFIKATOR tabident[MAX_POCET_IDENTIFIKATORU];
int pocet_identifikatoru = 0;


int najdi_identifikator(char *nazev)
{
        int i;
        for (i=pocet_identifikatoru-1; i>=0; i--)
        {
                if (strcmp(tabident[i].nazev, nazev) == 0) break;
        }
        if (i == -1) return CHYBA;
        else return i;
}


int pridej_identifikator(char *nazev)
{
        if (najdi_identifikator(nazev) == CHYBA)
        {
                strncpy(tabident[pocet_identifikatoru].nazev, nazev, MAX_DELKA_IDENTIFIKATORU);
                pocet_identifikatoru++;
                return OK;
        }
        else return CHYBA;
}
%}
```

Žádné jiné funkce potřebovat nebudeme. Ještě musíme definovat typ sémantické hodnoty tokenu rozpoznaného lexikálním analyzátorem - v případě čísel `int`, v případě identifikátorů řetězec. K tomu se používá konstrukce `%union` a některé přepínače:

```
%locations
%pure-parser
%union {
 int cislo;
 char *retezec;
}

%token  CISLO
%token  IDENTIFIKATOR
```

Krom čísel a identifikátorů budeme potřebovat tokeny pro klíčová slova, operátory a další hlouposti. Sémantické hodnoty u nich nepotřebujeme, naproti tomu musíme definovat priority kvůli jednoznačné syntaktické analýze.

```
%token K_VAR K_BEGIN K_END
%token O_PLUS O_MINUS O_KRAT O_DELENO
%token PRIRAD STREDNIK TECKA CARKA ZAVORKA_L ZAVORKA_P

%left O_PLUS O_MINUS
%left O_KRAT O_DELENO
%left UMINUS
%%
```

Ze záhlaví je to vše, můžeme se pustit do gramatiky samotného jazyka. Program (session) se skládá z definice proměnných a jednoho příkazu:

```
session:        vardecl command TECKA
                ;
```

Definice proměnných je buď prázdná (tj. nemáme žádné proměnné), nebo je složena z několika příkazů `var` následovaných seznamem proměnných ukončeným středníkem. Najdeme-li nějaký název identifikátoru, hned ho pošleme do tabulky identifikátorů.

```
vardecl:         /* prazdny */
                | vardecl K_VAR varlist STREDNIK
                ;
varlist:        IDENTIFIKATOR                   {       pridej_identifikator($1);
                                                }
                | varlist CARKA IDENTIFIKATOR   {       pridej_identifikator($3);
                                                }
                ;
```

Příkaz je jednoduché přiřazení, nebo blok `begin-end` s několika příkazy oddělenými středníky, nebo může být prázdný.

```
command:        assignment
                | K_BEGIN commandlist K_END
                |
                ;
```

```
commandlist:    /* prazdny */
                | commandlist command STREDNIK
                ;
```

Přiřazení je syntakticky jednoduché, musíme se ale poprat se sémantikou, resp. výkonnou částí. Gramatiku zkonstruujeme tak, že symbol `expression` vygeneruje potřebné instrukce k vyčíslení výrazu. Na vrcholu zásobníku pak bude hodnota, kterou zapíšeme do místa vyhrazeného přiřazované proměnné. Pro jednoduchost budeme předpokládat, že adresa tohoto místa je rovna číslu identifikátoru v tabulce identifikátorů (což není správně, ale zjednoduší nám to tuto verzi překladače).

```
assignment:     IDENTIFIKATOR PRIRAD expression {       int i;
                                                        i = najdi_identifikator($1);
                                                        if (i==CHYBA)
                                                        {
                                                                printf("Neznamy identifikator\n");
                                                        }
                                                        gen(STO, 0, i);
                                                }
                ;
```

Syntaxe aní sémantika samotného výrazu už by neměla překvapit.

```
expression:     CISLO                           {       gen(LIT, 0, $1); }
                | IDENTIFIKATOR                 {       int i;
                                                        i = najdi_identifikator($1);
                                                        if (i==CHYBA)
                                                        {
                                                                printf("Neznamy identifikator\n");
                                                        }
                                                        gen(LOD, 0, i);
                                                }
                | expression O_PLUS expression  {       gen(OPR, 0, ADD); }
                | expression O_MINUS expression {       gen(OPR, 0, SUB); }
                | expression O_KRAT expression  {       gen(OPR, 0, MUL); }
                | expression O_DELENO expression        {       gen(OPR, 0, DIV); }
                | O_MINUS expression %prec UMINUS       {       gen(OPR, 0, NEG); }
                | ZAVORKA_L expression ZAVORKA_P        { }
                ;
%%
```

Konečně napíšeme obligátní main a další nezbytné věci.

```
int main(int argc, char **argv)
{
  yyparse();
  vypis_kod();
  return 0;
}

int yyerror(char *s)
{
  printf("error: %s\n", s);
}

int yywrap(void)
{
  return 1;
}
```

A je to! Chcete-li, stáhněte si celý zdroják.

Rozšíření o příkaz if-then

Při překladu podmíněného příkazu je situace trochu komplikovanější; po rozpoznání klíčového slova IF a překladu podmínky potřebujeme generovat podmíněný skok, kterým se případně přeskočí část za klíčovým slovem THEN. Její délku ale budeme znát až časem; někam si proto potřebujeme schovat adresu oné instrukce s nepodmíněným skokem a až to bude možné, dodatečně ji modifikovat. Musíme uvažovat i vnořené příkazy IF-THEN; proto bude nejlepším úložištěm takové informace zásobník. Předchozí příklad proto v záhlaví doplníme jednoduchou implementací zásobníku a pak doplníme tu a tam další řádky.

```
#define MAX_DELKA_ZASOBNIKU     100
int zasobnik[MAX_DELKA_ZASOBNIKU];
int vrchol;

void push(int i)
{
        vrchol++;
        if (vrchol >= MAX_DELKA_ZASOBNIKU)
        {
                fprintf(stderr, "Preteceni zasobniku\n");
                exit(0);
        }
        zasobnik[vrchol] = i;
}


int pop(void)
{
        int i;
        if (vrchol<0)
        {
                fprintf(stderr, "Podteceni zasobniku\n");
                exit(0);
        }
        i = zasobnik[vrchol];
        vrchol--;
        return i;
}
```

K rozpoznání podmíněného příkazu využijeme nové tokeny. Jelikož jsou jednoduché, nebudeme si explicitně ukazovat nový lexikální analyzátor.

```
%token K_IF K_THEN
%token O_EQ O_NE O_LT O_LE O_GT O_GE
```

V minulém příkladu jsme ani pořádně neukončili program instrukcí RET. To napravíme pochopitelně v definici startovacího symbolu gramatiky. Na tomto místě bude dobré si připomenout, že na generování instrukce RET se překladač dostane v okamžiku, kdy rozpoznal celý program, tedy vygeneroval všechny instrukce.

```
session:        vardecl command TECKA           {       gen(RET, 0, 0); }
                ;
```

Musíme pochopitelně rozšířit definici příkazu:

```
command:        assignment
                | conditional
                | K_BEGIN commandlist K_END
                |
                ;
```

Symbol conditional je náš podmíněný příkaz. Vypadá tak, že začíná klíčovým slovem `if` a je následován podmínkou. Symbol condition je zodpovědný za její rozpoznání a vygenerování příslušných instrukcí. Jakmile se tedy analyzátor dostane za něj, může vygenerovat instrukci podmíněného skoku (jejíž adresu si musí zapamatovat v zásobníku). Pak bude pokračovat ověřením přítomnosti klíčového slova `then`. Symbol command, který je za ním, zařídí překlad then-větve do instrukcí. Jakmile bude hotov, bude zřejmé, kam skákat v případě, že podmínka za if nebyla splněna - na instrukci, která bude vygenerována na konec stávajícího kódu.

```
conditional:    K_IF condition                  {       push(delka_kodu);
                                                        gen(JMC, 0, 0); }
                K_THEN command                  {       int i;
                                                        i = pop();
                                                        kod[i][2] = delka_kodu; }
                ;
```

Samotné vyjádření podmínky je již triviální:

```
condition:      expression O_EQ expression      {       gen(OPR, 0, EQ); }
                | expression O_NE expression    {       gen(OPR, 0, NE); }
                | expression O_LT expression    {       gen(OPR, 0, LT); }
                | expression O_LE expression    {       gen(OPR, 0, LE); }
                | expression O_GT expression    {       gen(OPR, 0, GT); }
                | expression O_GE expression    {       gen(OPR, 0, GE); }
                ;
```

Chcete-li celý zdroják najednou, zde je.

Rozšíření o cyklus while

Vše potřebné pro přidání cyklu už vlastně máme hotovo. Přídáme jen token pro klíčová slova while a do (triviální) a rozšíříme možnosti příkazu:

```
command:        assignment
                | conditional
                | whileloop
                | K_BEGIN commandlist K_END
                |
                ;

whileloop:      K_WHILE                         {       push(delka_kodu);  /* zacatek kodu podminky */ }
                condition                       {       push(delka_kodu);
                                                        gen(JMC, 0, 0); } /* konec kodu podminky, resp.
                                                                             adresa instrukce podmíneného skoku
                                                                             na konec těla cyklu */
                K_DO command                    {       int zacatek, konec;
                                                        konec = pop();
                                                        zacatek = pop();
                                                        gen(JMC, 0, zacatek);
                                                        kod[konec][2] = delka_kodu; }
                ;
```

Tady si musíme pečlivě rozmyslet, co kdy do zásobníku vložit a kdy z něj vybrat. Čtenář si jistě kód analyzuje sám. A obligátně celý kód.

Rozšíření poslední, o procedury

Tím, že rozšíříme syntaxi o precedury, musíme přehodnotit i syntaktické chápání celého programu. Ten se bude nyní brát jako blok kódu. Takový blok může obsahovat definici proměnných a procedur, rozhodně musí obsahovat výkonnou část čili příkaz či konstrukci begin-end s několika příkazy. Blok hlavního programu bude ukončen tečkou. To je rozdíl oproti proceduře, která bude začínat klíčovým slovem procedure následovaným názvem procedury, středníkem, blokem kódu a koncovým středníkem. Nejprve si pro přehlednost uvedeme čistě syntaktický zápis, překladem se budeme zabývat až poté.

```
session:        block TECKA
                ;
block:          vardecl
                procedures
                command
                ;
procedures:     /* prazdny */
                | procedures
                  K_PROCEDURE IDENTIFIKATOR STREDNIK
                  block STREDNIK
                ;
```

Nyní začneme předělávat funkcionalitu překladače. Začněme infrastrukturou: během překladu si budeme muset pamatovat, jak vnořenou proceduru zpracováváme. Také si budeme muset pamatovat, kolik buněk v zásobníku si musíme v rámci bloku zarezervovat.

```
int uroven;
int adresa_v_bloku;
```

Tabulku symbolů také musíme rozšířit o typ identifikátoru, jeho úroveň zanoření (globální, v proceduře, v proceduře v proceduře atd.), délku (má smysl u procedur) a adresu (začátek kódu u procedur, v zásobníku u proměnných).

```
#define TYP_PROMENNA    0
#define TYP_PROCEDURA   1
char *nazev_typu[2] = {"promenna ", "procedura"};
typedef struct {
        char nazev[MAX_DELKA_IDENTIFIKATORU+1];
        int typ;
        int uroven;
        int adresa;
        int delka;
} T_IDENTIFIKATOR;
```

Tím pochopitelně musíme změnit funkce `najdi_identifikator` a `pridej_identifikator`. U funkce `najdi_identifikator` musíme tabulku prohledávat od konce, abychom našli definici pokud možno lokální. U této funkce navíc můžeme prohledávání tabulky omezit do určité úrovně; 0 znamená globální (čili prohledáváme celou tabulku od aktuální až do globální úrovně). Toho využijeme při přidávání identifikátoru, kdy umožníme překrýt identifikátor definovaný v jiné úrovni, nikoliv ale na úrovni aktuální.

```
int najdi_identifikator(char *nazev, int uroven)
{
        int i;
        for (i=pocet_identifikatoru-1; i>=0; i--)
        {
                if (tabident[i].uroven < uroven) { i=-1; break; }
                if (strcmp(tabident[i].nazev, nazev) == 0) break;
        }
        if (i == -1) return CHYBA;
        else return i;
}


int pridej_identifikator(char *nazev, int typ, int uroven, int adresa, int delka)
{
        if (najdi_identifikator(nazev, uroven) == CHYBA)
        {
                strncpy(tabident[pocet_identifikatoru].nazev, nazev, MAX_DELKA_IDENTIFIKATORU);
                tabident[pocet_identifikatoru].typ = typ;
                tabident[pocet_identifikatoru].uroven = uroven;
                tabident[pocet_identifikatoru].adresa = adresa;
                tabident[pocet_identifikatoru].delka = delka;
                pocet_identifikatoru++;
                return OK;
        }
        else return CHYBA;
}
```

Zřejmě budou zapotřebí nové tokeny:

```
%token K_PROCEDURE K_CALL
```

Konečně se dostáváme k vlastnímu překladu. Před začátkem bloku (hlavního programu nebo procedury) musíme generovat skok, který přeskočí případný kód vnořených procedur. Adresu této instrukce si zapamatujeme.

```
session:                                                {       uroven = 0;
                                                                delka_kodu = 0;
                                                                push(delka_kodu);
                                                                gen(JMP, 0, 0); }
                block TECKA
                ;
```

Ještě před začátkem analýzy bloku řekneme pomocí proměnná `adresa_v_bloku`, že potřebujeme alokovat 3 buňky v zásobníku. Tato proměnná může změnit svou hodnotu při vyhodnocování definic proměnných. Jakmile zpracujeme i vnořené procedury, víme, kde bude začínat kód bloku - skok, který jsme si vygenerovali v symbolu session (případně v procedure, viz dále), můžeme upravit na aktuálně generovanou instrukci. Touto instrukcí bude INT a zajistíme jí alokaci správného počtu buněk v zásobníku. Jakmile zpracujeme symbolem command i tělo bloku, můžeme generovat instrukci RET pro návrat z procedury, případně z hlavního programu.
Podmíněný příkaz `if (uroven > 0)` si vysvětlíme za chvíli.

```
block:                                                  {       adresa_v_bloku = 3; }
                vardecl procedures                      {       int i;
                                                                i = pop();
                                                                kod[i][2] = delka_kodu;
                                                                if (uroven > 0)
                                                                {
                                                                        i = pop();
                                                                        tabident[i].adresa = delka_kodu;
                                                                }
                                                                gen(INT, 0, adresa_v_bloku);
                                                        }
                command                                 {       gen(RET, 0, 0); }
                ;
```

Seznam procedur může být prázdný, případně obsahuje klíčové slovo `procedure`, název procedury, středník, blok příkazů a středník (a další takto definované procedury). Při čtení následujícího kódu je zajímavé všimnout si, že po zpracování hlavičky procedury ukládáme do zásobníku čtyři hodnoty, po zpracování bloku ale vybíráme pouze dvě. To proto, že třetí a čtvrtá je vybrána při samotném zpracování bloku (viz výše). Při zpracování bloku totiž potřebujeme přepsat instrukci skoku JMP generovanou symbolem procedures a zapsat do tabulky symbolů k právě definované proceduře začátek jejího kódu.

```
procedures:     /* prazdny */
                | procedures
                  K_PROCEDURE IDENTIFIKATOR STREDNIK    {       pridej_identifikator($3, TYP_PROCEDURA, uroven, delka_kodu, 0);
                                                                uroven++;
```

```
                                                push(pocet_identifikatoru);   /* pro obnoveni po zpracovani
                                                                                 podprocedur /*
                                                push(adresa_v_bloku);         /* pro obnoveni po zpracovani
                                                                                 podprocedur /*
                                                push(pocet_identifikatoru-1); /* posledni identifikator zarazeny
                                                                                 do tabulky identifikatoru,
                                                                                 tj. nazev procedury; vyuzije
                                                                                 ho symbol block /*
                                                push(delka_kodu);             /* adresa instrukce JMP, vyuzije ji
                                                                                 symbol block */
                                                gen(JMP, 0, 0); }
                  block STREDNIK                {       adresa_v_bloku = pop();
                                                        pocet_identifikatoru = pop();
                                                        tabident[pocet_identifikatoru-1].delka =
                                                          delka_kodu - tabident[pocet_identifikatoru-1].adresa;
                                                        uroven--; }
                ;
```

Rozšíříme množinu příkazů o volání procedury:

```
command:        assignment
                | conditional
                | whileloop
                | callprocedure
                | K_BEGIN commandlist K_END
                |
                ;
```

A nakonec doplníme správné volání procedury. Současně si musíme uvědomit, že úroveň identifikátoru se promítla i do manipulace s proměnnými, tedy je třeba upravit i příslušnou část symbolu expression.

```
callprocedure:  K_CALL IDENTIFIKATOR            {       int i;
                                                        i = najdi_identifikator($2, 0);
                                                        if (i==CHYBA)
                                                        {
                                                                printf("Neznamy identifikator\n");
                                                        }
                                                        gen(CAL, uroven-tabident[i].uroven, tabident[i].adresa); }
expression:     ...
                | IDENTIFIKATOR                 {       int i;
                                                        i = najdi_identifikator($1, 0);
                                                        if (i == CHYBA)
                                                        {
                                                                printf("Neznamy identifikator\n");
                                                        }
                                                        gen(LOD, uroven - tabident[i].uroven, tabident[i].adresa);
                                                }
                ...
```

A to je vše! Překladač je hotový (zdrojáky zde), do plné krásy mu zbývá jen přidělat konstanty. To je přenecháno čtenáři coby cvičení.

Domácí úkol

Poslední verze překladače rozšiřte o vyhodnocování podmínek o operátory `AND`, `OR` a `NOT` s obvyklými prioritami. Při překladu do instrukcí využívejte jen stávajících, tj. nedefinujte nové `OPR 0, ?`, ale využijte to, co již existuje. Příklad akceptovatelné konstrukce:

```
if a < 3 and b < 4 or not (x > 3 or y > 5) then ....
```

Poslední změna: 19.11.2013