Zadatak 1

Izmijenite zapis gramatike naredbi u oblik:

```
stmt: null_stmt
      | VARIABLE '=' expr  null_stmt
     | PRINT expr null_stmt
     | WHILE expr null_stmt stmt_list END
      | DO stmt_list UNTIL  expr  null_stmt
     | IF expr null_stmt stmt_list else_part
       ;
```

Sada gramatika sadrži naredbu za do-until petlju i u uvjetnim izrazima više nije potrebno pisati zagrade.

Izvršite odgovarajuće promjene u parseru, i testirajte intepreter u interaktivnom modu i s test programom.

```
a=5;b=2; c= 0;
do
   c = c + a
   if(c > 10)
      c = 10
   end
 b = b + 1
until b > 5
print c
```

Koristite ELL parser generator.

Datoteka kak.grm

U datoteci **kak.grm** treba dodati izraz za prepoznavanje DO – UNTIL petlje.

```
program: (stmt)* end;
end : EXIT | T_EOF;
stmt: null_stmt
      | VARIABLE '=' expr  null_stmt
      | PRINT expr null_stmt
      | WHILE expr null_stmt stmt_list END
      | DO stmt_list UNTIL expr null_stmt
      | IF expr null_stmt stmt_list else_part
            ;
else_part :  END  |  ELSE stmt_list END;
null_stmt: ';' | NL ;
stmt_list: stmt  (stmt)*;
expr : add_expr (relop add_expr)?;
add_expr :  ( '+' | '-' )? term  ( addop term)*;
term : expfactor (mulop expfactor)*;
expfactor :  factor ('^' factor)?;
factor : NUMBER
        | VARIABLE
        | MATFUN '(' expr ')'
        | '(' expr ')'
        ;
relop  :  EQ | '<' | '>' | LE | GE | NE ;
addop : '+' | '-' ;
mulop : '*' | '/' ;
```

ELL Parser

Nakon izmjene gramatike, u command promptu se pomoću naredbe **ell kak.grm** generiraju datoteke **kak.grm_par.c** i **kak.grm_par.h** koje se zatim preimenuju u datoteke **kakPar.c** i **kakPar.h** redom. Navedene datoteke se dodaju u VS projekt.

Datoteka kakPar.h

```
#ifndef INC_PARSER_H
#define INC_PARSER_H

#ifdef USE_LEX
      extern char *yytext;
      int yylex();
      #define getToken  yylex
      #define lexeme  yytext
#else
      int getToken();
      extern char *lexeme;
#endif

extern int lookahead;

#define EXIT        260
#define T_EOF       261
#define VARIABLE    263
```

```
#define PRINT       266
#define WHILE       267
#define END         269
#define DO          270
#define UNTIL       271
#define IF          272
#define ELSE        274
#define NL          276
#define NUMBER      293
#define MATFUN      294
#define EQ          297
#define LE          300
#define GE          301
#define NE          302

void program();
void stmt();
void end();
void null_stmt();
void expr();
void stmt_list();
void else_part();
void add_expr();
void relop();
void term();
void addop();
void expfactor();
void mulop();
void factor();

#endif
```

Datoteka kakPar.c

```
#include <stdio.h>
#include "kak.grm_par.h"

int lookahead;

void Error(int code)
{
       printf("Error:%d\n",code);
       exit(1);
}

void match(int t)
{
        if(lookahead == t) lookahead = getToken();
        else Error(-1);
}

void program()
{
       switch (lookahead) {
       case EXIT: case T_EOF: case VARIABLE: case PRINT: case WHILE: case DO: case IF:
case ';': case NL:
              while (lookahead == VARIABLE || lookahead == PRINT || lookahead == WHILE
|| lookahead == DO || lookahead == IF || lookahead == ';' || lookahead == NL ) {
                           stmt();
              }
              end();
```

```
		break;

	default:
		Error(0);
	}
}

void stmt()
{
	switch (lookahead) {
	case ';': case NL:
		null_stmt(); break;

	case VARIABLE:
		match(VARIABLE);
		match('=');
		expr();
		null_stmt();
		break;

	case PRINT:
		match(PRINT);
		expr();
		null_stmt();
		break;

	case WHILE:
		match(WHILE);
		expr();
		null_stmt();
		stmt_list();
		match(END);
		break;

	case DO:
		match(DO);
		stmt_list();
		match(UNTIL);
		expr();
		null_stmt();
		break;

	case IF:
		match(IF);
		expr();
		null_stmt();
		stmt_list();
		else_part();
		break;

	default:
		Error(1);
	}
}

void end()
{
	switch (lookahead) {
	case EXIT:
		match(EXIT); break;
	case T_EOF:
		match(T_EOF); break;
```

```
	default:
		Error(2);
	}
}

void null_stmt()
{
	switch (lookahead) {
	case ';':
		match(';'); break;

	case NL:
		match(NL); break;

	default:
		Error(3);
	}
}

void expr()
{
	switch (lookahead) {
	case VARIABLE: case '+': case '-': case NUMBER: case MATFUN: case '(':
		add_expr();
		if (lookahead == EQ || lookahead == '<' || lookahead == '>' || lookahead
== LE || lookahead == GE || lookahead == NE ) {
				relop();
				add_expr();
		}
		break;

	default:
		Error(4);
	}
}

void stmt_list()
{
	switch (lookahead) {
	case VARIABLE: case PRINT: case WHILE: case DO: case IF: case ';': case NL:
		stmt();
		while (lookahead == VARIABLE || lookahead == PRINT || lookahead == WHILE
|| lookahead == DO || lookahead == IF || lookahead == ';' || lookahead == NL ) {
				stmt();
		}
		break;

	default:
		Error(5);
	}
}

void else_part()
{
	switch (lookahead) {
	case END:
		match(END); break;

	case ELSE:
		match(ELSE);
		stmt_list();
```

```
		match(END);
		break;

	default:
		Error(6);
	}
}

void add_expr()
{
	switch (lookahead) {
	case VARIABLE: case '+': case '-': case NUMBER: case MATFUN: case '(':
		if (lookahead == '+' || lookahead == '-' ) {
			if (lookahead == '+' ) {
				match('+'); }
			else if (lookahead == '-' ) {
				match('-'); }
		}
		term();
		while (lookahead == '+' || lookahead == '-' ) {
				addop();
				term();
		}
		break;

	default:
		Error(7);
	}
}

void relop()
{
	switch (lookahead) {
	case EQ:
		match(EQ); break;

	case '<':
		match('<'); break;

	case '>':
		match('>'); break;

	case LE:
		match(LE); break;

	case GE:
		match(GE); break;

	case NE:
		match(NE); break;

	default:
		Error(8);
	}
}

void term()
{
	switch (lookahead) {
	case VARIABLE: case NUMBER: case MATFUN: case '(':
		expfactor();
		while (lookahead == '*' || lookahead == '/' ) {
```

```
                        mulop();
                        expfactor();
                }
                break;

        default:
                Error(9);
        }
}

void addop()
{
        switch (lookahead) {
        case '+':
                match('+'); break;

        case '-':
                match('-'); break;

        default:
                Error(10);
        }
}

void expfactor()
{
        switch (lookahead) {
        case VARIABLE: case NUMBER: case MATFUN: case '(':
                factor();
                if (lookahead == '^' ) {
                        match('^');
                        factor();
                }
                break;

        default:
                Error(11);
        }
}

void mulop()
{
        switch (lookahead) {
        case '*':
                match('*'); break;

        case '/':
                match('/'); break;

        default:
                Error(12);
        }
}

void factor()
{
        switch (lookahead) {
        case NUMBER:
                match(NUMBER);
                break;

        case VARIABLE:
```

```
                match(VARIABLE);
                break;

        case MATFUN:
                match(MATFUN);
                match('(');
                expr();
                match(')');
                break;

        case '(':
                match('(');
                expr();
                match(')');
                break;

        default:
                Error(13);
        }
}
```

VS Projekt

Datoteke **kak.h** i **kakSymbol.c** ostaju nepromijenjene. U datoteku **kak.c** doda se izvršavanje naredbi za operand ***DO*** u funkciji **Execute (nodeT *n)**. Potrebno je dodati akcije koje se izvršavaju prilikom prepoznavanja ključnih riječi „do“ i „until“ u funkciji **getToken (void)** koja se nalazi u datoteci **kakLex.c.**

Datoteka kak.c

```
#include <math.h>
#include <stdio.h>
#include "kak.h"
#include "kakPar.h"

NodeT *Opr3(int oper, NodeT * n1, NodeT * n2, NodeT * n3)
{
        NodeT *n = xmalloc(sizeof(NodeT));
        n->kind = kindOpr;
        n->oper = oper;
        n->left  = n1;
        n->right = n2;
        n->next  = n3;
        return n;
}

NodeT *Opr2(int oper, NodeT * n1, NodeT * n2) {
        return Opr3(oper, n1, n2, NULL);
}

NodeT *Opr1(int oper, NodeT * n1) {
        return Opr3(oper, n1, NULL, NULL);
}

NodeT *NumConst(double value)
{
        NodeT *n = xmalloc(sizeof(NodeT));
```

```
	n->kind = kindNum;
	n->value = value;
	return n;
}

NodeT *Var(Symbol *sp)
{
	NodeT *n = xmalloc(sizeof(NodeT));
	n->kind = kindVar;
	n->sp = sp;
	return n;
}

NodeT *MatFun(pmatFunT pmatFun, NodeT *expr)
{
	NodeT *n = Opr3(0, expr, NULL, NULL);
	n->kind = kindMatFun;
	n->pmatFun = pmatFun;
	return n;
}

NodeT *AppendStmt(NodeT * list, NodeT * stmt)
{
	NodeT *n = Opr1(STMT_LIST, stmt);
	if(!list)
		list = n;
	else {
		NodeT * p = list;
		while(p->next != NULL) p = p->next;
		p->next= n;
	}
	return list;
}

void freeNode(NodeT *n) {
	if (!n) return;
	if (n->kind == kindOpr) {
		freeNode(n->left);
		freeNode(n->right);
		freeNode(n->next);
	}
	free (n);
}

void exec_error(char *str)
{
  printf("%s\n",str);
  exit(0);
}

double Execute(NodeT *n)
{
	if (!n) return 0;
	if(n->kind == kindNum)  return n->value;
	else if(n->kind ==  kindMatFun)   return (*n->pmatFun)(Execute(n->left));
	else if(n->kind == kindVar) {
		if(!n->sp){
			exec_error("Variable not defined\n");
			return 0;
		}
		return n->sp->val;
	}
```

```
	else if(n->kind == kindOpr)switch(n->oper)
	{
		case WHILE:	while(Execute(n->left)) Execute(n->right);
				return 0;

		case DO:		Execute(n->left);
					while(!Execute(n->right)) Execute(n->left);
					return 0;

		case IF:	if (Execute(n->left))  Execute(n->right);
					else if (n->next)	Execute(n->next);
					return 0;

		case PRINT:	printf("> %g\n", Execute(n->left) ); return 0;

		case STMT_LIST: do {Execute(n->left); n=n->next; }while(n); return 0;

		case '=':	return n->left->sp->val = Execute(n->right);

		case '+':	return Execute(n->left) + Execute(n->right);
		case '-':	return Execute(n->left) - Execute(n->right);
		case '*':	return Execute(n->left) * Execute(n->right);
		case '/':	{ double op2 = Execute(n->right);
				if(op2 == 0){ exec_error("Zero devide\n"); op2=1;}
				return Execute(n->left) / op2;
					}
		case '^':	{ double op2 = Execute(n->right);
					return pow(Execute(n->left), op2); }

		case '<':	return Execute(n->left) < Execute(n->right);
		case '>':	return Execute(n->left) > Execute(n->right);
		case GE:	return Execute(n->left) >= Execute(n->right);
		case LE:	return Execute(n->left) <= Execute(n->right);
		case NE:	return Execute(n->left) != Execute(n->right);
		case EQ:	return Execute(n->left) == Execute(n->right);
		default:
				exec_error("Bad operator\n");
				return 0;
	}

	return 0;}

jmp_buf   jumpdata;

int main(int argc, char *argv[])
{
	if(argc == 2) {
		input = fopen(argv[1], "rt");
		if (input == NULL) {
			printf ("Ne moze otvoriti datoteku: %s", argv[1]);
			exit(1);
		}
	}
	else
		input = stdin;

	setjmp(jumpdata);
	lookahead = getToken();
	program();
	return 0;
}
```

Datoteka kakLex.c

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "Kak.h"
#include "KakPar.h"

#define LEX_MAX_SIZE 32

char lexeme[LEX_MAX_SIZE + 1];
int lineno=1;
static int pos=0;
FILE * input;

void syn_error(char *str) {
  printf("Line %d, Pos %d: %s\n",lineno,pos,str);
  exit(1);
}

static struct pFunStrut {
       char *name;
       pmatFunT pmatFun;
}

pFunArr[] =  { {"sin", sin},  {"cos", cos},  {"asin", asin},  {"acos", acos},
  {"tan", tan},  {"atan", atan}, {"sinh", sinh},  {"cosh", cosh},
  {"exp", exp},  {"abs", fabs},  {"log", log},  {"log10", log10},
  {"sqrt", sqrt}, {"ceil", ceil}, {"floor", floor},
  {NULL, NULL}
};

pmatFunT getFunPtr(char *name) {
  int i = 0;
  while(pFunArr[i].name != NULL) {
     if(strcmp(pFunArr[i].name, name) == 0)
              return pFunArr[i].pmatFun;
     i++;
  }
  return NULL;
}

int getToken(void)
{
    int ch=getc(input);
       pos++;

       while (ch ==' ' || ch =='\t' || ch =='\r') { ch=getc(input); pos++; }

       if (ch =='\n') { lineno++; pos=1;
              if(input == stdin) ungetc(';', stdin);
       return NL;}

       if (ch =='?') { pos=1; return PRINT;}

       if (ch == '=') { ch=getc(input); if (ch == '=') {pos++; return EQ;} else
{ungetc(ch, input); return '=';}}
       if (ch == '>') { ch=getc(input); if (ch == '=') {pos++; return GE;} else
{ungetc(ch, input); return '>';}}
       if (ch == '<') { ch=getc(input); if (ch == '=') {pos++; return LE;} else
{ungetc(ch, input); return '<';}}
```

```
	if (ch == '!') { ch=getc(input); if (ch == '=') {pos++; return NE;} else
{ungetc(ch, input); return '!';}}
	if (ch == EOF) return EXIT;

	if (isdigit(ch))
	{
		int i=0;
		while (isdigit(ch))  {
			lexeme[i]=ch;
			ch=getc(input); pos++; i++;
			if (i > LEX_MAX_SIZE) syn_error ("Buffer overflow");
		}
		if(ch != '.'){
			lexeme[i]= '\0';
			if (ch != EOF){	ungetc(ch,input);	pos--;}
			return NUMBER;
		}
		else {
			 lexeme[i]='.';
			 ch = getc(input); pos++; i++;
			 while (isdigit(ch)) {
			lexeme[i]= ch;
			ch = getc(input); pos++; i++;
			if (i > LEX_MAX_SIZE) syn_error ("Buffer overflow");
			 }
			 lexeme[i]='\0';
			 if (ch != EOF){ ungetc(ch, input); pos--; }

			 return NUMBER;
		}
	}
	else if (isalpha(ch))
	{
		int i=0;
		while (isalnum(ch)) {
			lexeme[i]= tolower(ch);
			ch=getc(input); pos++; i++;
			if (i > LEX_MAX_SIZE) syn_error ("Buffer overflow");
		}
		lexeme[i]= '\0';

		if (ch != EOF)  {ungetc(ch, input); pos--; }

		if	(!strcmp(lexeme, "while")) return WHILE;
		else if (!strcmp(lexeme, "print")) return PRINT;
		else if (!strcmp(lexeme, "if"))   return IF;
		else if (!strcmp(lexeme, "do"))   return DO;
		else if (!strcmp(lexeme, "until")) return UNTIL;
		else if (!strcmp(lexeme, "end"))  return END;
		else if (!strcmp(lexeme, "exit")) return EXIT;
		else if (!strcmp(lexeme, "pi")) {
			strcpy(lexeme, "3.14159265358979323846");
			return NUMBER;
		}
		else if (getFunPtr(lexeme)) return MATFUN;
		else return VARIABLE;
    }
    else {
       return ch;
    }
}
```

Testiranje programa

Program **kak.exe** je testiran u interaktivnom modu. Primjer testiranja:

```
> kak.exe
x=5;
do  x=x+5;
until x < 12
print x

> 15
```

Zadatak 2

Napišite kod leksičkog analizatora pomoću programa Lex. Vodite računa o tome da leksički analizator prihvaća realni broj u eksponencijalnom obliku:

```
DOT \.
DIGIT [0-9]
%%

({DIGIT}+{DOT}?) | ({DIGIT}*{DOT}{DIGIT}+)) ([eE][-+]?[0-9]+)?
```

Također implementirajte svojstvo da leksički analizator tretira kao komentar (i odbaci) sve znakove koji slijede iza znaka ' , sve do kraja linije.

Napišite prikladnu specifikaciju leksičkog analizatora i pomoću programa LEX (ili flex) napravite datoteku lex.yy.c koja sadrži funkciju yylex() koja vraća token.

Lex

U specifikaciju leksičkog analizatora dodano je prepoznavanje realnog broja u običnom i eksponencijalnom obliku te prepoznavanje znaka za komentar do kraja linije.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "kakPar.h"
%}

DOT          \.
DIGIT        [0-9]
EXP          [eE][-+]?{DIGIT}+

%%
[ \t\r]+                  ;
'.*\n                     ;
({DIGIT}+{DOT}?)|({DIGIT}*{DOT}{DIGIT}+){EXP}?     return NUMBER;
exit                      return EXIT;
<<EOF>>                   return T_EOF;
[a-zA-Z][a-zA-Z0-9]*      return VARIABLE;
print                     return PRINT;
while                     return WHILE;
do                        return DO;
until                     return UNTIL;
if                        return IF;
;                         return ';';
\n                        return NL;
\^                        return '^';
\+                        return '+';
\-                        return '-';
\*                        return '*';
\/                        return '/';
\=\=                      return EQ;
\<                        return '<';
\>                        return '>';
\<\=                      return LE;
\>\=                      return GE;
\!\=                      return NE;
sin | cos | asin | acos | tan | atan | sinh | cosh | exp | abs | log | log10 |
sqrt | ceil | floor      return MATFUN;

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