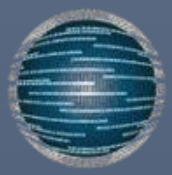

CEID

Αρχές Γλωσσών Προγραμματισμού & Μεταφραστών

Εργαστηριακή Άσκηση 2021

Αβραμόπουλος Μιχαήλ
AM: 1067451

Καραγιάννης Αλέξανδρος
AM: 1067452

Πανταζάρας Ευστράτιος
AM: 1067490

Περιεχόμενα

Ανάλυση 2

BNF 2

Παραδοχές 6

Σχόλια – Παραδείγματα 6

Ερώτημα 1 6

Ερώτημα 2 9

Ερώτημα 3 10

Ερώτημα 4 13

Κώδικας 15

Flex 15

Bison 18

Ανάλυση

BNF

```
<digit>             ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<digits>            ::= <digit> | <digit> <digits>
<letter>            ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
                      | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
                      | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
                      | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
                      | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
                      | "y" | "z"
<underscore>        ::= "_"
<symbol>            ::= "|" | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+"
                      | "," | "-" | "." | "/" | ":" | ";" | ">" | "=" | "<" | "?"
                      | "@" | "[" | "]" | "^" | "`" | "{" | "}" | "~" | """ | "_"
<lu>                ::= <letter> | "_"
<ldu>               ::= <letter> | <digit> | "_"
<ldusy>             ::= <ldu> | <symbol>
<ldusys>            ::= <ldusy> | <ldusy> <ldusys>
<ldus>              ::= <ldu> | <ldu> <ldus>
<varname>           ::= <lu> <ldus>
<PROGRAM>           ::= "PROGRAM " <varname> "\n"
<STRUCT>            ::= "STRUCT " <varname> "\n"
<COMP>              ::= "==" | "!=" | ">" | "<"
<character>         ::= "'" <ldusy> "'"
<multiline_comment> ::= "/*"
<string>            ::= """<ldusys>"""
```

```
<comment>           ::= "%" <ldusys> "\n"
<file>              ::= <PROGRAM> <structs> <functions> <main>
                      | <PROGRAM> <functions> <main>
                      | <PROGRAM> <structs> <main>
                      | <PROGRAM> <main>
<structs>           ::= <struct>
                      | <struct> <structs>
<declarations>      ::= "VARS" <declaration>
                      | <declarations> <declaration>
<struct>            ::= <STRUCT> <declarations> "ENDSTRUCT"
                      | "TYPEDEF" "STRUCT" <declarations> <varname> "ENDSTRUCT"
<main>              ::= "STARTMAIN" <declarations> <block> "ENDMAIN"
                      | "STARTMAIN" <declarations> "ENDMAIN"
                      | "STARTMAIN" <block> "ENDMAIN"
                      | "STARTMAIN" "ENDMAIN"
block               ::= <statement>
                      | <statement> <block>
<statement>         ::= <print>
                      | <assignment>
                      | <if>
                      | <for>
                      | <while>
                      | <switch>
                      | <break>
                      | <invocation> ";"
                      | <return>
<print>             ::= "PRINT" "(" <string> ")" ";"
                      | "PRINT" "(" <string> "," <print_var_list> ")" ";"
```

```
<invocation>        ::= <varname> "(" <parameter_list> ")"
<parameter_list>    ::= <expression>
                      | <expression> "," <parameter_list>
<declaration>       ::= <var_type> <var_list> ";"
<assignment>        ::= <varname> "=" <expression> ";"
                      | <array> "=" <expression> ";"
<expression>        ::= <digits>
                      | <array>
                      | <character>
                      | <varname>
                      | <expression> <operator> <expression>
                      | "(" <expression> ")"
                      | <invocation>
<var_list>          ::= <varname>
                      | <array>
                      | <var_list> "," <varname>
                      | <var_list> "," <array>
<print_var_list>    ::= <varname>
                      | <array>
                      | <var_list> "," <varname>
                      | <var_list> "," <array>
<operator>          ::= "+"
                      | "-"
                      | "*"
                      | "/"
                      | "^"
<array>             ::= <varname> "[" <digits> "]"
                      | <varname> "[" <varname> "]"
<functions>         ::= <function>
                      | <function> <functions>
```

```
<function>          ::= "FUNCTION" <varname> "(" <param_declaration> ")" <declarations> <block> "END_FUNCTION"
                    | "FUNCTION" <varname> "(" <param_declaration> ")" <declarations> "END_FUNCTION"
                    | "FUNCTION" <varname> "(" <param_declaration> ")" "END_FUNCTION"
                    | "FUNCTION" <varname> "(" <param_declaration> ")" <block> "END_FUNCTION"
param_declaration   ::= <var_type> <varname>
                    | <var_type> <varname> "," <param_declaration>
<var_type>          ::= "CHAR"
                    | "INTEGER"
                    | <varname>
<return>            ::= "RETURN" <expression> ";"
<break>             ::= "BREAK" ";"
<bool_op>           ::= "AND"
                    | "OR"
condition           ::= <expression> <COMP> <expression>
                    | <condition> <bool_op> <condition>
<while>             ::= WHILE "(" <condition> ")" <block> "ENDWHILE"
<if>                ::= "IF" "(" <condition> ")" "THEN" <block> "ENDIF"
                    | "IF" "(" <condition> ")" "THEN" <block> "ELSE" <block> "ENDIF"
                    | "IF" "(" <condition> ")" "THEN" <block> <elseifs> "ENDIF"
                    | "IF" "(" <condition> ")" "THEN" <block> <elseifs> "ELSE" <block> "ENDIF"
<elseifs>           ::= "ELSEIF" "(" <condition> ")" <block>
                    | "ELSEIF" "(" <condition> ")" <block> <elseifs>
<cases>             ::= "CASE" "(" <expression> ")" ":" <block>
                    | "CASE" "(" <expression> ")" ":" <block> <cases>
<switch>            ::= "SWITCH" "(" <expression> ")" <cases> "ENDSWITCH"
                    | "SWITCH" "(" <expression> ")" <cases> "DEFAULT" ":" <block> "ENDSWITCH"
<for>               ::= "FOR" <varname> ":" "=" <digits> "TO" <digits> "STEP" <digits> <block> "ENDFOR"
                    | "FOR" <varname> ":" "=" <digits> "TO" <digits> "STEP" <digits> "ENDFOR"
```

Παραδοχές

- Μπορούμε να έχουμε πεπερασμένο αριθμό μεταβλητών / συναρτήσεων / struct – που ρυθμίζεται ως μήκος μερικών arrays στο bison, στα πλαίσια του project αυτό το νούμερο είναι 256
- Εάν οριστεί νέος τύπος δεδομένων χρήστη με 'STRUCT' αυτός μπορεί να χρησιμοποιηθεί παρακάτω ΜΟΝΟ αν έχει οριστεί με τη σύνταξη 'TYPEDEF STRUCT' (όμως το όνομα του νέου τύπου είναι αυθαίρετο)
- Μια μεταβλητή που έχει οριστεί οπουδήποτε στο πρόγραμμα δεν μπορεί να προκαλέσει `undeclared identifier error`. Για παράδειγμα αν ορίσουμε την μεταβλητή Α τοπικά σε μια συνάρτηση και μετά έξω από αυτήν την συνάρτηση την καλέσουμε, δεν θα προκύψει `undeclared identifier error`. Αυτό έγινε για να αποφύγουμε την περίπλοκη υλοποίηση scopes που θα απαιτούσαν αναδρομικό / περίπλοκο αλγόριθμο

Σχόλια – Παραδείγματα

Ερώτημα 1

Αρχικά υλοποιήσαμε την βασική μορφή της γλώσσας στους κανόνες flex και bison (που φαίνονται στο παραπάνω BNF και στον παρακάτω κώδικα). Για να κάνουμε συντακτική και λεκτική ανάλυση σε ολόκληρο αρχείο και όχι γραμμή-γραμμή από το stdin κάνουμε redirect την είσοδο των flex και bison στο όνομα του αρχείου που παίρνουμε σαν command line input. «Αδειάζουμε» επίσης τα περιεχόμενα του αρχείου στο stdout με τη βοήθεια της catFile:

```
void catFile(char* filename){
    FILE * fp = fopen(filename, "r");
    char next = fgetc(fp);
    while (next != EOF)
    {
        printf ("%c", next);
        next = fgetc(fp);
    }
    fclose(fp);
    printf("\n");
}

int main(int argc, char **argv){
    // print file contents
    catFile(argv[1]);

    // redirect input to file
    yyin = fopen(argv[1],"r");
    yyparse();

    return 0;
}
```

Μετά τα compile τρέχουμε τον αναλυτή μας ως εξής:

```
Alex@alex-desktop ~
$ ./myParser.exe input.c
```

Αλλάζουμε και την yyerror προκειμένου να δείχνει σε ποια γραμμή υπάρχει λάθος:

```
void yyerror(char *s) {
    fprintf(stderr, "at line %d: %s\n", yylineno, s);
}
```

Η yylineno ορίζεται ως extern παραπάνω και για να είναι σωστά ορισμένη πρέπει να δώσουμε το %option yylineno στο αρχείο του lex.

Περνάμε τώρα ως είσοδο ένα δοκιμαστικό πρόγραμμα που είναι συντακτικά σωστό και δοκιμάζει πολλούς από τους κανόνες της γλώσσας μας:

```
Alex@alex-desktop ~
$ ./ourlang.exe input.c
PROGRAM my_program
    FUNCTION mean(INTEGER in1, INTEGER in2)
        VARS
        INTEGER output;
        output = (in1+in2)/2;
        RETURN output;
    END_FUNCTION

    FUNCTION printArray(CHAR a[4])
        VARS
        INTEGER i;

        FOR i := 1 TO 4 STEP 1
            PRINT("a = ", a[i]);
        ENDFOR

        RETURN 0;
    END_FUNCTION

    STARTMAIN
        VARS
        INTEGER num1, num2, result;
        CHAR letters[4];

        num1 = 2;
        num2 = 6;
        letters[0] = 'a';
        letters[1] = 'b';
        letters[2] = 'c';
        letters[3] = 'd';

        result = mean(num1, num2);

        printArray(letters);
    ENDMAIN
```

Όπως βλέπουμε ο αναλυτής δεν βρίσκει κανένα λάθος. Αν προκαλέσουμε εσκεμμένα ένα λάθος πχ ξεχνάμε το ; στη γραμμή letters[1]=’b’; τότε βλέπουμε πως εντοπίζει την γραμμή του λάθους:

```
Alex@alex-desktop ~
$ ./ourlang.exe input.c
PROGRAM my_program
    FUNCTION mean(INTEGER in1, INTEGER in2)
        VARS
        INTEGER output;
        output = (in1+in2)/2;
        RETURN output;
    END_FUNCTION

    FUNCTION printArray(CHAR a[4])
        VARS
        INTEGER i;

        FOR i := 1 TO 4 STEP 1
            PRINT("a = ", a[i]);
        ENDFOR

        RETURN 0;
    END_FUNCTION

    STARTMAIN
        VARS
        INTEGER num1, num2, result;
        CHAR letters[4];

        num1 = 2;
        num2 = 6;
        letters[0] = 'a';
        letters[1] = 'b'
        letters[2] = 'c';
        letters[3] = 'd';

        result = mean(num1, num2);

        printArray(letters);
    ENDMAIN
at line 29: syntax error
```

```
28      letters[1] = 'b'
29      letters[2] = 'c';
```

(Θεωρεί πως η γραμμή 29 είναι προέκταση της 28 καθώς δεν τελειώνει με (;))

Ερώτημα 2

Προσθέτουμε τη δυνατότητα ορισμού structs (τύπων δεδομένου χρήστη) μέσα στα οποία επιτρέπεται μόνο να ορίσουμε μεταβλητές. Αν δοκιμάσουμε σε μια είσοδο:

```
$ ./myParser.exe input.c
PROGRAM my_program
    STRUCT complexNumber
        VARS
        INTEGER real, imaginary;
    ENDSTRUCT

    STRUCT customString
        VARS
        INTEGER char[256];
    ENDSTRUCT

    STARTMAIN
        VARS
        INTEGER num;

        num = 2;
        PRINT("num = ", num);
    ENDMAIN
```

Όπως βλέπουμε ο parser δεν εντοπίζει λάθος στην είσοδο. Αν δοκιμάσουμε να κάνουμε πχ PRINT στο σώμα του struct (συντακτικό λάθος):

```
$ ./myParser.exe input.c
PROGRAM my_program
    STRUCT complexNumber
        VARS
        INTEGER real, imaginary;

        PRINT("hello world");
    ENDSTRUCT

    STRUCT customString
        VARS
        CHAR chars[256];
    ENDSTRUCT

    STARTMAIN
        VARS
        INTEGER num;

        num = 2;
        PRINT("num = ", num);
    ENDMAIN
at line 6: syntax error
```

Βλέπουμε πως εντοπίζει το λάθος στη γραμμή του PRINT.

Αν χρησιμοποιήσουμε την σύνταξη με το TYPEDEF, το νέο αυτό είδος μεταβλητής χρήστη μπορεί να χρησιμοποιηθεί όπως τα κανονικά:

```
$ bison -d test.y &&  lex test.l && gcc lex.yy.c test.tab.c -o
& ./myParser.exe input.c
test.y: warning: 7 shift/reduce conflicts [-Wconflicts-sr]
PROGRAM my_program
    TYPEDEF STRUCT complexNumber
        VARS
        INTEGER real, imaginary;
    complexNumber ENDSTRUCT

    TYPEDEF STRUCT customString
        VARS
        CHAR chars[256];
    customString ENDSTRUCT

    STARTMAIN
        VARS
        complexNumber cmplx;
        customString my_str1;

        PRINT("complex num = ", cmplx);
        PRINT("custom string = ", my_str1);
    ENDMAIN
```

Ερώτημα 3

Υλοποιούμε πίνακες πεπερασμένου μεγέθους που αποθηκεύουν όλες τις μεταβλητές, συναρτήσεις, structs και τύπους δεδομένων χρήστη. Έτσι σε κάθε αναφορά μεταβλητής (επιλεγμένους κανόνες του bison) ελέγχουμε αν η μεταβλητή έχει ορισθεί προηγουμένως: (εδώ δείχνουμε για παράδειγμα μόνο τις συναρτήσεις)

```
// functions
char funcs[256][256] = {0};
void addFunc(char *);
int funcExists(char *);
```

Ο πίνακας funcs αποθηκεύει όλες τις ορισμένες συναρτήσεις και χωράει έως 256 διαφορετικά ονόματα συναρτήσεων το κάθε ένα έως 256 χαρακτήρες σε μήκος, και αρχικοποιείται ολόκληρος με ASCII 0. Η συνάρτηση addFunc παίρνει ως είσοδο ένα νέο όνομα συνάρτησης και το προσθέτει στο πίνακα funcs:

```
void addFunc(char * newFunc){
    int exists = 0;

    // find first empty id
    int i = 0;
    while(funcs[i][0] != 0){
        i++;
        if(strcmp(funcs[i], newFunc) == 0){
            exists = 1;
        }
    }

    // only add it if it doesnt already exist
    if(exists==0){
        // add id
        int j = 0;
        while(newFunc[j] != '\0'){
            funcs[i][j] = newFunc[j];
            j++;
        }
    }
}
```

Ελέγχει πρώτα αν υπάρχει ήδη το ίδιο όνομα και αν όχι, τότε το προσθέτει. Τέλος η συνάρτηση funcExists παίρνει ως είσοδο ένα όνομα συνάρτησης και ελέγχει αν έχει οριστεί προηγουμένως (αν υπάρχει μέσα στον funcs):

```
int funcExists(char * func){
    int i = 0;
    while(funcs[i][0] != 0){
        if(strcmp(funcs[i], func) == 0){
            return 1;
        }
        i++;
    }

    return 0;
}
```

Αυτός ο πίνακας και οι δύο συναρτήσεις καλύπτουν την αναγνώριση των συναρτήσεων μόνο, αντίστοιχα υλοποιούμε για τις μεταβλητές, τα struct και τους τύπους δεδομένων χρήστη.

Έτσι οι κανόνες του bison που ορίζουν νέα ονόματα μοιάζουν με τον παρακάτω:

```
param_declaration: var_type VARNAME                              {addId($<stringValue>2);}
                 | var_type array                                {addId($<stringValue>2);}
                 | var_type array ',' param_declaration          {addId($<stringValue>2);}
                 | var_type VARNAME ',' param_declaration        {addId($<stringValue>2);}
                 ;
```

(κάθε νέο variable name προστίθεται στη λίστα των ορισμένων με τη βοήθεια της addId)

ενώ οι κανόνες που χρειάζεται να ελέγξουν αν έχουν οριστεί μεταβλητές μοιάζουν με:

```
expression: DIGITS                          {;}
          | array                           {if(idExists($<stringValue>1) == 0){undeclaredError($<stringValue>1);YYERROR;}}
          | CHARACTER                       {;}
          | VARNAME                         {if(idExists($<stringValue>1) == 0){undeclaredError($<stringValue>1);YYERROR;}}
          | expression operator expression  {;}
          | '(' expression ')'              {;}
          | invocation                      {;}
          ;
```

(εάν η idExists επιστρέψει 0 τότε προκαλούμε error με την YYERROR και εκτυπώνουμε custom error message με την συνάρτηση undeclaredError).

Σε μερικές περιπτώσεις χρειάζεται να «ανεβάσουμε» το string value μιας μεταβλητής σε «παραπάνω» κανόνα, οπότε χρησιμοποιούμε την σύνταξη με τα $:

```
array: VARNAME '[' DIGITS ']'       {$<stringValue>$ = $<stringValue>1;}
```

Επιστρέφοντας στο παράδειγμα, δοκιμάζουμε να χρησιμοποιήσουμε μη ορισμένες μεταβλητές:

```
PROGRAM my_program
    STRUCT complexNumber
        VARS
        INTEGER real, imaginary;
    ENDSTRUCT

    STARTMAIN
        VARS
        INTEGER a, b, c;

        d = a+b+c;
        PRINT("num = ", num);
    ENDMAIN
at line 11: undeclared identifier: d
```

```
PROGRAM my_program
    STARTMAIN
        VARS
        INTEGER a, b, c;

        a = mean(b, c);
    ENDMAIN
at line 6: undeclared identifier: mean
```

Ο parser μας εντοπίζει τα λάθη και εκτυπώνει το άγνωστο όνομα.

Ερώτημα 4

Για τα σχόλια πολλαπλών γραμμών αφού δεν επιτρέπεται η χρήση regex χρησιμοποιήσαμε συνάρτηση μέσα στο λεξικό αναλυτή που όταν εντοπίζει τους χαρακτήρες /* παραλείπει όλες τις επόμενες γραμμές μέχρι να βρει */:

```
int skipMultilineComment(){
    int nextc=input();

    while (nextc!=EOF){
        if ((char)nextc == '*'){
            nextc=input();
            if ((char)nextc=='/') return 1;
            else continue;
            break;
        }
        nextc=input();
    }

    return 0;
}
```

Έτσι ο κανόνες του lex γίνεται:

```
"/*"                    {if(!skipMultilineComment()) return -1;}
```

Αν η skipMultilineComment επιστρέψει 0, δηλαδή βρει EOF πριν τους χαρακτήρες */ τότε ο κανόνας lex επιστρέφει -1, που θεωρείται κατευθείαν error.

```
PROGRAM my_program
    /* comment start

    STARTMAIN
        VARS
        INTEGER a, b, c;

        a = mean(b, c);
    ENDMAIN

    (no end?)

at line 12: syntax error
```

Δείχνουμε και ένα παράδειγμα με σωστά single line και multiline σχόλια:

```
PROGRAM my_program
    /*
        this program is
        only a test
    */

    /*
        no rights reserved
    */
    FUNCTION mean(INTEGER x, INTEGER y)
        RETURN (x+y)/2;
    END_FUNCTION

    STARTMAIN
        VARS
        INTEGER a, b, c;

        % calculate the mean of two numbers
        a = mean(b, c);
    ENDMAIN
```

Κώδικας

Flex

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

int skipMultilineComment();
void substring(char* in, char* out, int start, int end);
%}
%option noyywrap
%option yylineno

%%
"PROGRAM "[a-zA-Z_][a-zA-Z0-9_]*\n    {return PROGRAM;}
"FUNCTION"                            {return FUNCTION;}
"VARS"                                {return VARS;}
"CHAR"                                {return CHAR;}
"INTEGER"                             {return INTEGER;}
"END_FUNCTION"                        {return END_FUNCTION;}
"RETURN"                              {return RETURN;}
"STARTMAIN"                           {return STARTMAIN;}
"ENDMAIN"                             {return ENDMAIN;}
"IF"                                  {return IF;}
"THEN"                                {return THEN;}
"ELSE"                                {return ELSE;}
```

```
"ELSEIF"                                {return ELSEIF;}
"ENDIF"                                 {return ENDIF;}
"WHILE"                                         {return WHILE;}
"ENDWHILE"                              {return ENDWHILE;}
"FOR"                                   {return FOR;}
"TO"                                    {return TO;}
"STEP"                                  {return STEP;}
"ENDFOR"                                {return ENDFOR;}
"BREAK"                                         {return BREAK;}
"PRINT"                                 {return PRINT;}
"SWITCH"                                {return SWITCH;}
"CASE"                                  {return CASE;}
"DEFAULT"                               {return DEFAULT;}
"ENDSWITCH"                             {return ENDSWITCH;}
"AND"                                   {return AND;}
"OR"                                    {return OR;}
"STRUCT "[a-zA-Z_][a-zA-Z0-9_]*\n           {yylval.stringValue=strdup(yytext);return STRUCT;}
"ENDSTRUCT"                             {return ENDSTRUCT;}
"TYPEDEF"                               {return TYPEDEF;}
[0-9]+                                          {yylval.intValue= atoi(yytext); return DIGITS;}
"'"."'"                                 {yylval.stringValue= strdup(yytext); return CHARACTER;}
"=="|"!="|">"|"<"       {return COMP;}
"/*"                {if(!skipMultilineComment())return -1;}
[+*-/^,=():%]                           {return yytext[0];}
";"                                             {return SEMICOLON;}
\"(\\.|[^\\"])*\"       {return STRING;}
"%".*\n                 ;
[ \t\n]                                 ;
[a-zA-Z_][a-zA-Z0-9_]*   {yylval.stringValue=strdup(yytext);return VARNAME;}
```

```
"["                 {return yytext[0];}
"]"                 {return yytext[0];}

%%

int skipMultilineComment(){
   int nextc=input();

   while (nextc!=0){
      if ((char)nextc == '*'){
         nextc=input();
         if ((char)nextc=='/') return 1;
         if (nextc==0) continue;
      }
      nextc=input();
   }

   return 0;
}

void substring(char* out, char* in, int start, int end){
   int length = end - start;
   strncpy(out,in+start,length);
   out[length] = 0;
}
```

Bison

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int yylex();
void yyerror (char *);
extern FILE * yyin;
extern int * yylineno;
extern void substring(char*, char*, int, int);
// identifiers
char ids[256][256] = {0};
void addId(char *);
int idExists(char *);
// functions
char funcs[256][256] = {0};
void addFunc(char *);
int funcExists(char *);
// structs
char structs[256][256] = {0};
void addStruct(char *);
int structExists(char *);
// type names
char typeNames[256][256] = {0};
void addTypeName(char *);
int typeNameExists(char *);
```

```
void undeclaredError(char * id);
%}

%union
{
   int intValue;
   char *stringValue;
}

%token DIGITS
%token CHARACTER
%token PROGRAM
%token FUNCTION
%token VARS
%token CHAR
%token INTEGER
%token END_FUNCTION
%token RETURN
%token STARTMAIN
%token ENDMAIN
%token IF
%token THEN
%token ELSEIF
%token ENDIF
%token WHILE
%token ENDWHILE
%token FOR
%token TO
%token STEP
```

```
%token ENDFOR
%token BREAK
%token PRINT
%token SWITCH
%token CASE
%token DEFAULT
%token ENDSWITCH
%token AND
%token OR
%token VARNAME
%token SEMICOLON
%token STRING
%token COMP
%token ELSE
%token STRUCT
%token ENDSTRUCT
%token TYPEDEF

%start file

%%

file: PROGRAM structs functions main    {;}
        | PROGRAM functions main                {;}
        | PROGRAM structs main                  {;}
   | PROGRAM main                               {;}
        ;

structs: struct
```

```
        | struct structs
        ;

declarations: VARS declaration
                | declarations declaration
                ;
struct: STRUCT declarations ENDSTRUCT{char newId[256]; substring(newId, $<stringValue>1, 7, strlen($<stringValue>1)-1); addStruct(newId);}
        | TYPEDEF STRUCT declarations VARNAME ENDSTRUCT {char newId[256]; substring(newId, $<stringValue>2, 7, strlen($<stringValue>2)-1);if(strcmp(newId, $<stringValue>4) == 0){addStruct(newId);addTypeName(newId);}else{yyerror("TYPEDEF STRUCT name different than ENDSTRUCT name!");YYERROR;}}
        ;

main: STARTMAIN declarations block ENDMAIN      {;}
        | STARTMAIN declarations ENDMAIN        {;}
        | STARTMAIN block ENDMAIN               {;}
        | STARTMAIN ENDMAIN                     {;}
        ;

block: statement                {$<intValue>$ = $<intValue>1;}
        | statement block       {$<intValue>$ = $<intValue>1 || $<intValue>2;}
        ;

statement: print                {$<intValue>$ = 0;}
        | assignment            {$<intValue>$ = 0;}
        | if                    {$<intValue>$ = 0;}
        | for                   {$<intValue>$ = 0;}
        | while                 {$<intValue>$ = 0;}
        | switch                {$<intValue>$ = 0;}
```

```
                | break                                   {$<intValue>$ = 0;}
                | invocation SEMICOLON                  {$<intValue>$ = 0;}
                | return                                        {$<intValue>$ = 1;}
                ;

print: PRINT '(' STRING ')' SEMICOLON                                                   {;}
        | PRINT '(' STRING ',' print_var_list ')' SEMICOLON             {;}
        ;

invocation: VARNAME '(' parameter_list ')'              {if(funcExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}

parameter_list: expression                                              {;}
                        | expression ',' parameter_list {;}
                        ;

declaration: var_type var_list SEMICOLON                {;}

assignment: VARNAME '=' expression SEMICOLON            {if(idExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}
                | array '=' expression SEMICOLON                {if(idExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}
                ;

expression: DIGITS                                              {;}
                | array                                         {if(idExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}
                | CHARACTER                                     {;}
                | VARNAME                                       {if(idExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}
```

```
                    | expression operator expression          {;}
                    | '(' expression ')'                                {;}
                    | invocation                                                {;}
                    ;

var_list: VARNAME                                       {addId($<stringValue>1);}
                    | array                                             {addId($<stringValue>1);}
      | var_list ',' VARNAME                {addId($<stringValue>3);}
                    | var_list ',' array                {addId($<stringValue>3);}
                    ;

print_var_list: VARNAME                                                 {if(idExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}
                        | array
          {if(idExists($<stringValue>1) == 0){undeclaredError($<stringValue>1);YYERROR;}}
                        | var_list ',' VARNAME                  {if(idExists($<stringValue>3) ==
0){undeclaredError($<stringValue>3);YYERROR;}}
                        | var_list ',' array                    {if(idExists($<stringValue>3) ==
0){undeclaredError($<stringValue>3);YYERROR;}}
                        ;

operator: '+'               {;}
                    | '-'               {;}
                    | '*'               {;}
                    | '/'               {;}
                    | '^'               {;}
                    ;

array: VARNAME '[' DIGITS ']'           {$<stringValue>$ = $<stringValue>1;}
```

```
        | VARNAME '[' VARNAME ']'          {$<stringValue>$ = $<stringValue>1;
if(idExists($<stringValue>3) == 0){undeclaredError($<stringValue>3);YYERROR;}}

        ;


functions: function                      {;}

         | function functions   {;}

         ;


function: FUNCTION VARNAME '(' param_declaration ')' declarations block END_FUNCTION
        {if($<intValue>7 == 0){yyerror("function does not
return!");YYERROR;}addFunc($<stringValue>2);}

         | FUNCTION VARNAME '(' param_declaration ')' declarations END_FUNCTION
         {yyerror("function does not return!");YYERROR;}

         | FUNCTION VARNAME '(' param_declaration ')' END_FUNCTION
                  {yyerror("empty functions are not allowed!");YYERROR;}

         | FUNCTION VARNAME '(' param_declaration ')' block END_FUNCTION
         {if($<intValue>6 == 0){yyerror("function does not
return!");YYERROR;}addFunc($<stringValue>2);}

         ;


param_declaration: var_type VARNAME
        {addId($<stringValue>2);}

                  | var_type array
        {addId($<stringValue>2);}

                  | var_type array ',' param_declaration
        {addId($<stringValue>2);}

                  | var_type VARNAME ',' param_declaration
        {addId($<stringValue>2);}

                  ;


var_type: CHAR           {;}

         | INTEGER       {;}
```

```
                    | VARNAME                 {if(typeNameExists($<stringValue>1) ==
0){undeclaredError($<stringValue>1);YYERROR;}}
                    ;


return: RETURN expression SEMICOLON             {;}
    ;


break: BREAK SEMICOLON                      {;}
          ;


bool_op: AND              {;}
           | OR                       {;}
           ;


condition: expression COMP expression                       {;}
                    | condition bool_op condition                       {;}
                    ;


while: WHILE '(' condition ')' block ENDWHILE     {;}
          ;


if: IF '(' condition ')' THEN block ENDIF                                                   {;}
  | IF '(' condition ')' THEN block ELSE block ENDIF                                {;}
  | IF '(' condition ')' THEN block elseifs ENDIF                                   {;}
  | IF '(' condition ')' THEN block elseifs ELSE block ENDIF            {;}
 ;


elseifs: ELSEIF '(' condition ')' block                                             {;}
           | ELSEIF '(' condition ')' block elseifs                          {;}
```

```
            ;

cases: CASE '(' expression ')' ':' block                    {;}
            | CASE '(' expression ')' ':' block cases       {;}
            ;

switch: SWITCH '(' expression ')' cases ENDSWITCH
            | SWITCH '(' expression ')' cases DEFAULT ':' block ENDSWITCH
            ;

for: FOR VARNAME ':' '=' DIGITS TO DIGITS STEP DIGITS block ENDFOR
  | FOR VARNAME ':' '=' DIGITS TO DIGITS STEP DIGITS ENDFOR
  ;

%%

void yyerror(char *s) {
  fprintf(stderr, "at line %d: %s\n", yylineno, s);
}

void catFile(char* filename){
          FILE * fp = fopen(filename, "r");
  char next = fgetc(fp);
  while (next != EOF)
  {
     printf ("%c", next);
     next = fgetc(fp);
  }
  fclose(fp);
```

```
        printf("\n");
}

int main(int argc, char **argv){

        // print file contents
        catFile(argv[1]);

        // redirect input to file
        yyin = fopen(argv[1],"r");
        yyparse();

        return 0;
}

void addId(char * newId){
        int exists = 0;

        // find first empty id
        int i = 0;
        while(ids[i][0] != 0){
                i++;
                if(strcmp(ids[i], newId) == 0){
                        exists = 1;
                }
        }

        // only add it if it doesnt already exist
        if(exists==0){
```

```
                // add id
                int j = 0;
                while(newId[j] != '\0'){
                        ids[i][j] = newId[j];
                        j++;
                }
        }
}

int idExists(char * id){
        int i = 0;
        while(ids[i][0] != 0){
                if(strcmp(ids[i], id) == 0){
                        return 1;
                }
                i++;
        }

        return 0;
}

void addFunc(char * newFunc){
        int exists = 0;

        // find first empty id
        int i = 0;
        while(funcs[i][0] != 0){
                i++;
                if(strcmp(funcs[i], newFunc) == 0){
```

```
                exists = 1;
            }
        }

        // only add it if it doesnt already exist
        if(exists==0){
            // add id
            int j = 0;
            while(newFunc[j] != '\0'){
                funcs[i][j] = newFunc[j];
                j++;
            }
        }
}

int funcExists(char * func){
        int i = 0;
        while(funcs[i][0] != 0){
            if(strcmp(funcs[i], func) == 0){
                return 1;
            }
            i++;
        }

        return 0;
}

void addStruct(char * newStruct){
        int exists = 0;
```

```
	// find first empty id
	int i = 0;
	while(structs[i][0] != 0){
		i++;
		if(strcmp(structs[i], newStruct) == 0){
			exists = 1;
		}
	}

	// only add it if it doesnt already exist
	if(exists==0){
		// add id
		int j = 0;
		while(newStruct[j] != '\0'){
			structs[i][j] = newStruct[j];
			j++;
		}
	}
}

int structExists(char * struc){
	int i = 0;
	while(structs[i][0] != 0){
		if(strcmp(structs[i], struc) == 0){
			return 1;
		}
		i++;
	}
```

```
	return 0;
}

void addTypeName(char * newTypeName){
	int exists = 0;

	// find first empty id
	int i = 0;
	while(typeNames[i][0] != 0){
		i++;
		if(strcmp(typeNames[i], newTypeName) == 0){
			exists = 1;
		}
	}

	// only add it if it doesnt already exist
	if(exists==0){
		// add id
		int j = 0;
		while(newTypeName[j] != '\0'){
			typeNames[i][j] = newTypeName[j];
			j++;
		}
	}
}

int typeNameExists(char * typeName){
	int i = 0;
```

```
	while(typeNames[i][0] != 0){
		if(strcmp(typeNames[i], typeName) == 0){
			return 1;
		}
		i++;
	}

	return 0;
}

void undeclaredError(char * id){
	char message[270] = "undeclared identifier: ";
	yyerror(strcat(message, id));
}
```