2nd Programming Assignment Report

Vytaras Juraska

Compilers Principles and Design
Computer Science and Information Engineering student of
National Dong Hwa University

CONTENTS

1

PROBLEM DESCRIPTION

The problem description stated, that we needed to augment our T language parser with semantic actions for constructing parse trees. Also we needed to create a C code generator and make it a workable from T language to C compiler.

From the given files, we had already a partially completed:

- t_parse.y - bison file which required filling in semantic actions;
- t2c_tree.c - T to C generator, which required finishing expression generation, code generation and optionally an expression type analyzer.

Rest of the files were provided to us fully completed, including: t2c.c, t2c.h, t2c_tree.h, t_lex.l and 4 different test files.

2

PROGRAM WRITING HIGHLIGHTS

2.1 T_PARSE.Y

2.1.1 *astm*

```
astm      :       lID lASSIGN expr lSEMI
                  {
                    $$ = create_stm();
                    $$->stm_id = sASTM;
                    $$->exp1 = create_exp();
                    $$->exp1->exp_id = eASSIGN1;
                    strcpy( $$->exp1->name, $1 );
                    $$->exp1->exp1 = $3;
                  }
          ;
```

Assign Statement is considered to be a statement semantic, however it also includes an expression.

Here we start from creating the statement and giving it a pre-defined ID "sASTM", which all of them were defined in t2c_tree.h file - a file given to us which was fully completed. Considering whether a specific semantic was an expression or a statement, the name listing in t2c_tree.h file gave the hint. Some of them, like Write Statement included the same exact name of WSTM, however it had an expression and a statement defined.

Continuing forward, we need to create a statement and give it the name corresponding to it - "eASSIGN1". Next, we have an lID, hence we need can define the expressions name by relying on the lID. The line "strcpy($$->exp1->name, $1);" copies lID as a string to the expresions name variable, where $1 mean the first keyword in the description of "astm".

2.1.2 *wstm*

```
wstm      :       lWRITE lLP expr lCOMMA lQSTR lRP lSEMI
                  {
```

```
                $$ = create_stm();
                $$->stm_id = sWSTM;
                $$->exp1 = create_exp();
                $$->exp1->exp_id = eWSTM;
                $$->exp1->exp1 = $3;
                strcpy($$->exp1->qstr, $5);
                gen_rw[1][(ana_exptype($3)) - 1] = 1;

            }
    ;
```

This is the semantic action definition of "Write Statement". As referred before, it includes a statement and an expression, which were defined in the t2c_tree.h file.

In the C code generation file, writing and reading are defined by the types of variables, which are a subject of reading or writing operations. These operations are split into 3 types - integer, real and string.

Most of the actions defined here are the same as previous example, however here we are considering the expression itself and the string, which includes in the reading or writing operation. The string is copied to the "qstr" variable. Lastly, the "gen_rw" is called out, which is responsible for defining the reading writing function in the c code corresponding to what action is taken and what type of expression for that operation is required.

"ana_exptype" is a function which determines what kind of type that expression has. In the "gen_rw", the first array determines the operation, [1] being the operation of writing and [2] is reading. The next array depends on the type of expression.

2.2 T2C_TREE.C

2.2.1 *gen_exp eWSTM*

```
  case eWSTM:
int tmpwstm = ana_exptype( p );
if (tmpwstm == 1) {
  fprintf( yyout, "tiny_writeint(");
  gen_exp( p->exp1 );
  fprintf( yyout, ", %s);\n", p->qstr );
}
if (tmpwstm == 2) {
  fprintf( yyout, "tiny_writereal(");
  gen_exp( p->exp1 );
  fprintf( yyout, ", %s);\n", p->qstr );
}
if (tmpwstm == 3) {
```

```
    fprintf( yyout, "tiny_writestr(");
    gen_exp( p->exp1 );
    fprintf( yyout, ", &%s);\n", p->qstr );
  }
```

One of the examples for expression generation is the expression of writing statement. Here I also relied on the "ana_exptype" where I check what is the type of the expression used in the writing statement and according to that I defined the calling out function of "tiny_write".

After the type is identified, I print the beginning of function calling including left parenthesis, then I generate the expression, which is coming from semantics defined earlier. Finally I print the "qstr" string variable, separating expression and string with a comma, close parenthesis and end with a semicolon.

Most of the other expressions were pretty similar, there were some differences in each one of them, but I found writing and reading statements the most challenging, hence I refer to them on the program writing highlights.

2.2.2 *ana_exptype*

```
while (p->exp1) {
  if (p->next != NULL)
  {
    tEXP *nextP = p;
    while(nextP->next != NULL)
    {
      nextP = nextP->next;
      if (nextP->exp1->exp_id == 10)
      {
        tmp = tREAL; //real value used, expression is real
        break;
      }
    }
  }
  p=p->exp1;
}
```

This is a part of the expression type analyzer. This while function is responsible for going through the expressions and checking if the expression is a part of multiple expressions, or is it a simple variable definition.

We go through "exp1" of the expression until the expression in question does not have an "exp1" variable defined. During this check, we also consider that some expressions might have "next" variable, which would lead to further expressions. The biggest concern here is, as it was in one of the examples, that an expression can lead

to a mathematical equation, where more than one expression has to be checked to determine the type of expression.

In "test2.c" we have a writing operation call-out, where the expression of the call-out is defined as "2 * 3.14 * r". Here the variable "r" is a real type, "3.14" is also not an int type, however the first variable is "2", hence if we would just check the expression - wrongfully int type would be determined for this operation. To truly understand what type that would be, we have to consider other expressions, hence going through the "next" variable and looking if a real type variable has been used is crucial to understand the true type required to be used for this operation.

Does not matter, if the operation in the expression is addition or division, or multiplication, if a real type is used, that said expression is determined to also be a type of real.

the other part of the "ana_exptype" is a bit more simpler. It essentially includes a predefined function, which goes through syntax nodes defined by the given input and checks for the corresponding name of the expression, which gives the type for you. That would be used in most of the situations, where the expression is named and defined, and it does not have to be considered by looking at the equation use-case, but simply relying on the definition is enough to determine the type of expression.

3

PROGRAM LISTING

3.1 t_parse.y

```
    %{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include "t2c.h"
        #include "t2c_tree.h"
        #include "t_parse.h"
%}

%token lWRITE lREAD lIF lASSIGN
%token lRETURN lBEGIN lEND
%left  lEQU lNEQ lGT lLT lGE lLE
%left  lADD lMINUS
%left  lTIMES lDIVIDE
%token lLP lRP
%token lINT lREAL lSTRING
%token lELSE
%token lMAIN
%token lSEMI lCOMMA
%token lID lINUM lRNUM lQSTR

%union { tSTM* sm;
         tEXP* ex;
         int iv;
         float rv;
         char* sr;
       }

%type <sm> prog
%type <sm> mthdcls
%type <sm> mthdcl
```

```
%type <sm> block
%type <sm> stmts
%type <sm> stmt
%type <sm> vardcl
%type <sm> astm
%type <sm> rstm
%type <sm> istm
%type <sm> wstm
%type <sm> dstm

%type <ex> type
%type <ex> formals
%type <ex> formal
%type <ex> oformal
%type <ex> expr
%type <ex> mexprs
%type <ex> mexpr
%type <ex> pexprs
%type <ex> pexpr
%type <ex> bexpr
%type <ex> aparams
%type <ex> oparams

%type <sr> lID
%type <iv> lINUM
%type <rv> lRNUM
%type <sr> lQSTR

%expect 1

%%
prog    :       mthdcls
                { $$ = $1;
                  program = $$; }
        |
                { printf("****** Parsing failed!\n"); }
        ;

mthdcls :       mthdcl mthdcls
                { $$ = $1;
                  $$->next = $2; }
        |       mthdcl
                { $$ = $1; }
```

```
        ;

type    :       lINT
                { $$ = create_exp();
                  $$->exp_id = eTYPE;
                  printf("MyTiny parse: type ok!\n");
                  $$->ival = tINT; }
        |       lREAL
                { $$ = create_exp();
                  $$->exp_id = eTYPE;
                  $$->ival = tREAL; }
        ;

mthdcl  :       type lMAIN lID lLP formals lRP block
                { $$ = create_stm();
                  $$->stm_id = sMAIN;
                  $$->exp1 = create_exp();
                  $$->exp1->exp_id = eID;
                  strcpy( $$->exp1->name, $3 );
                  $$->exp1->exp1 = $1;
                  $$->exp2 = $5;
                  $$->stm1 = $7;
                  symtab = create_symnode( $3, $1->ival ); }
        |       type lID lLP formals lRP block
                { $$ = create_stm();
                  $$->stm_id = sMDCL;
                  $$->exp1 = create_exp();
                  $$->exp1->exp_id = eID;
                  strcpy( $$->exp1->name, $2 );
                  $$->exp1->exp1 = $1;
                  $$->exp2 = $4;
                  $$->stm1 = $6;
                  symtab = create_symnode( $2, $1->ival ); }
        ;

formals :       formal oformal
                { $$ = $1;
                  $$->next = $2; }
        |
                { $$ = NULL; }
        ;

formal  :       type lID
```

```
                { $$ = create_exp();
                  $$->exp_id = eFORM;
                  $$->exp1 = $1;
                  strcpy( $$->name, $2 ); }
        ;

oformal :       lCOMMA formal oformal
                { $$ = $2;
                  $$->next = $3; }
        |
                { $$ = NULL; }
        ;

block   :       lBEGIN stmts lEND
                { $$ = create_stm();
                  $$->stm_id = sBLOCK;
                  printf("MyTiny parse: block ok!\n");
                  $$->stm1 = $2; }
        ;

stmts   :       stmt stmts
                { $$ = $1;
                  $$->next = $2; }
        |       stmt
                { $$ = $1; }
        ;

stmt    :       block
                { $$ = $1; }
        |       vardcl
                { $$ = $1; }
        |       astm
                { $$ = $1; }
        |       rstm
                { $$ = $1; }
        |       istm
                { $$ = $1; }
        |       wstm
                { $$ = $1; }
        |       dstm
                { $$ = $1; }
        ;
```

```
vardcl  :       type lID lSEMI
                { // Done.
                  $$ = create_stm();
                  $$->stm_id = sVDCL1;
                  $$->exp1 = $1;
                  strcpy( $$->exp1->name, $2 );
                  symtab = create_symnode( $2, $1->ival );
                }
        |       type astm
                { // Done.
                  $$ = create_stm();
                  $$->stm_id = sVDCL2;
                  $$->exp1 = $1;
                  $$->stm1 = $2;
                  symtab = create_symnode( $2->exp1->name, $1->ival );
                }
        ;

astm    :       lID lASSIGN expr lSEMI
                { // Done.
                  $$ = create_stm();
                  $$->stm_id = sASTM;
                  $$->exp1 = create_exp();
                  $$->exp1->exp_id = eASSIGN1;
                  strcpy( $$->exp1->name, $1 );
                  $$->exp1->exp1 = $3;
                }
        ;

rstm    :       lRETURN expr lSEMI
                { $$ = create_stm();
                  $$->stm_id = sRSTM;
                  $$->exp1 = $2; }
        ;

istm    :       lIF lLP bexpr lRP stmt
                { // Done.
                  $$ = create_stm();
                  $$->stm_id = sISTM;
                  $$->exp1 = $3;
                  $$->stm1 = $5;
                }
        |       lIF lLP bexpr lRP stmt lELSE stmt
```

```
                { $$ = create_stm();
                  $$->stm_id = sISTM;
                  $$->exp1 = $3;
                  $$->stm1 = $5;
                  $$->stm2 = $7; }
        ;

wstm    :       lWRITE lLP expr lCOMMA lQSTR lRP lSEMI
                { // Done.
                  $$ = create_stm();
                  $$->stm_id = sWSTM;
                  $$->exp1 = create_exp();
                  $$->exp1->exp_id = eWSTM;
                  $$->exp1->exp1 = $3;
                  strcpy($$->exp1->qstr, $5);
                  gen_rw[1][(ana_exptype($3)) - 1] = 1;

                }
        ;

dstm    :       lREAD lLP lID lCOMMA lQSTR lRP lSEMI
                { // Done.
                  $$ = create_stm();
                  $$->stm_id = sDSTM;
                  $$->exp1 = create_exp();
                  $$->exp1->exp_id = eDSTM;
                  strcpy( $$->exp1->name, $3 );
                  strcpy( $$->exp1->qstr, $5 );
                  gen_rw[0][(ana_exptype($$->exp1)) - 1] = 1;
                }
        ;

expr    :       mexpr mexprs
                { $$ = create_exp();
                  $$->exp_id = eEXPR;
                  $$->exp1 = $1;
                  $$->next = $2; }
        ;

mexprs  :       lADD mexpr mexprs
                { $$ = create_exp();
                  $$->exp_id = eADD;
                  $$->exp1 = $2;
```

```
                $$->next = $3; }
        |       lMINUS mexpr mexprs
                { $$ = create_exp();
                  $$->exp_id = eMINUS;
                  $$->exp1 = $2;
                  $$->next = $3; }
        |
                { $$ = NULL; }
        ;

mexpr   :       pexpr pexprs
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eMEXP;
                  $$->exp1 = $1;
                  $$->next = $2;
                }
        ;

pexprs  :       lTIMES pexpr pexprs
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eTIMES;
                  $$->exp1 = $2;
                  $$->next = $3;
                }
        |       lDIVIDE pexpr pexprs
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eDIVIDE;
                  $$->exp1 = $2;
                  $$->next = $3;
                }
        |
                { $$ = NULL; }
        ;

pexpr   :       lINUM
                { $$ = create_exp();
                  $$->exp_id = eINUM;
                  $$->ival = $1; }
        |       lRNUM
                { $$ = create_exp();
```

```
                $$->exp_id = eRNUM;
                $$->rval = $1; }
        |       lID
                { $$ = create_exp();
                $$->exp_id = eID;
                strcpy( $$->name, $1 ); }
        |       lLP expr lRP
                { // Done.
                $$ = create_exp();
                $$->exp_id = ePAREN;
                $$->exp1 = $2;
                }
        |       lID lLP aparams lRP
                { // Done.
                $$ = create_exp();
                $$->exp_id = eFUNC;
                strcpy( $$->name, $1 );
                $$->exp1 = $3;
                }
        ;

bexpr   :       expr lEQU expr
                { // Done.
                $$ = create_exp();
                $$->exp_id = eEQ;
                $$->exp1 = $1;
                $$->next = $3;
                }
        |       expr lNEQ expr
                { // Done.
                $$ = create_exp();
                $$->exp_id = eNE;
                $$->exp1 = $1;
                $$->next = $3;
                }
        |       expr lGT expr
                { // Done.
                $$ = create_exp();
                $$->exp_id = eGT;
                $$->exp1 = $1;
                $$->next = $3;
                }
        |       expr lLT expr
```

```
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eLT;
                  $$->exp1 = $1;
                  $$->next = $3;
                }
        |       expr lGE expr
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eGE;
                  $$->exp1 = $1;
                  $$->next = $3;
                }
        |       expr lLE expr
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eLE;
                  $$->exp1 = $1;
                  $$->next = $3;
                }
        ;

aparams :       expr oparams
                { // Done.
                  $$ = create_exp();
                  $$->exp_id = eAPARM;
                  $$->exp1 = $1;
                  $$->next = $2;
                }
        |
                { // Done.
                  $$ = NULL;
                }
        ;

oparams :       lCOMMA expr oparams
                { // Done.
                  $$ = $2;
                  $$->next = $3;
                }
        |
                { // Done.
                  $$ = NULL;
```

```
                }
        ;

%%

int yyerror(char *s)
{
        printf("%s\n",s);
        return 1;
}
```

3.2 T2C_TREE.C

```
    #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "t2c.h"
#include "t2c_tree.h"

char c_types1[3][6] = {"int", "float", "char*"};
char c_types2[3][5] = {"int", "real", "str"};

tEXP* create_exp ( ) {
  tEXP* tmp;
  tmp = (struct t_exp *)malloc(sizeof(struct t_exp));
  if( tmp ) {
    tmp->exp_id = eMIN;
    tmp->name[0] = '\0';
    tmp->ival = 0;
    tmp->rval = 0.0;
    tmp->qstr[0] = '\0';
    tmp->exp1 = NULL;
    tmp->next = NULL;
  }
  return tmp;
}

tSTM* create_stm ( ) {
  tSTM* tmp;
  tmp = (struct t_stm *)malloc(sizeof(struct t_stm));
  if( tmp ) {
    tmp->stm_id = sMIN;
```

```
    tmp->exp1 = NULL;
    tmp->exp2 = NULL;
    tmp->stm1 = NULL;
    tmp->stm2 = NULL;
    tmp->next = NULL;
  }
  return tmp;
}

symNODE* create_symnode ( char* s, int t ) {
  symNODE* tmp;
  tmp = (struct sym_node *)malloc(sizeof(struct sym_node));
  if( tmp ) {
    strcpy( tmp->name, s );
    tmp->type = t;
    tmp->next = symtab;
  }
  return tmp;
}

void free_exp ( tEXP* p ) {
  if( p ) {
    if( p->exp1 ) free_exp( p->exp1 );
    if( p->next ) free_exp( p->next );
    free( p );
  }
}

void free_stm ( tSTM* p ) {
  if( p ) {
    if( p->exp1 ) free_exp( p->exp1 );
    if( p->exp2 ) free_exp( p->exp2 );
    if( p->stm1 ) free_stm( p->stm1 );
    if( p->stm2 ) free_stm( p->stm2 );
    if( p->next ) free_stm( p->next );
    free( p );
  }
}

void free_symnode ( symNODE* p ) {
  if( p ) {
    if( p->next ) free_symnode( p->next );
    free( p );
```

```
  }
}

void print_exp ( tEXP* p ) {
  tEXP* te;
  if( p ) {
    switch( p->exp_id ) {
    case eTYPE: printf("eTYPE: %d\n", p->ival);
                break;
    case eFORM: printf("eFORM: ");
                print_exp( p->exp1 );
                printf(" %s", p->name);
                if (p->next) {
                    printf(", ");
                    print_exp( p->next );
                }
                printf("(end of eFORM)\n");
                break;
    case eEXPR: printf("eEXPR: ");
                print_exp( p->exp1 );
                if (p->next) {
                    print_exp( p->next );
                }
                printf("(end of eEXPR)\n");
                break;
    case eADD:
      printf("eADD: ");
      print_exp( p->exp1 );
      if (p->next) {
          print_exp( p->next );
      }
      printf("(end of eADD)\n");
                break;
    case eMINUS:
      printf("eMINUS: ");
                  print_exp( p->exp1 );
                  if (p->next) {
                    print_exp( p->next );
                  }
                  printf("(end of eMINUS)\n");
                break;
    case eMEXP:
      printf("eMEXP: ");
```

```
  print_exp( p->exp1 );
  if (p->next) {
      print_exp( p->next );
  }
  printf("(end of eMEXP)\n");
            break;
case eTIMES:
  printf("eTIMES: ");
  print_exp( p->exp1 );
  if (p->next) {
      print_exp( p->next );
  }
  printf("(end of eTIMES)\n");
            break;
case eDIVIDE:
  printf("eDIVIDE: ");
  print_exp( p->exp1 );
  if (p->next) {
      print_exp( p->next );
  }
  printf("(end of eDIVIDE)\n");
            break;
case eINUM:
  printf("eINUM: %d\n", p->ival);
            break;
case eRNUM:
  printf("eRNUM: %f\n", p->rval);
            break;
case eID:
  printf("eID: %s\n", p->name);
            break;
case ePAREN:
  printf("(\n");
  print_exp( p->exp1 );
  printf(")\n");
            break;
case eFUNC:
  printf("eFUNC: %s(", p->name);
  print_exp( p->exp1 );
  printf(")(end of eFUNC)\n");
            break;
case eEQ:
  printf("eEQ: ");
```

```
      print_exp( p->exp1 );
      printf("(end of eEQ)\n");
                break;
    case eNE:
      printf("eNE: ");
      print_exp( p->exp1 );
      printf("(end of eNE)\n");
                break;
    case eAPARM:
      printf("eAPARM: ");
      print_exp( p->exp1 );
        if (p->next) print_exp( p->next );
      printf("(end of eAPRAM)\n");
                break;
    case eASSIGN1:
      printf("eASSIGN1: %s = ", p->name);
      print_exp( p->exp1 );
      printf("(end of eASSIGN1)\n");
                break;
    case eASSIGN2:
      printf("eASSIGN2: %s = %s\n", p->name, p->qstr);
                break;
    case eWSTM:
      printf("eWSTM: WRITE( ");
      print_exp( p->exp1 );
      printf(", %s )\n", p->qstr);
                break;
    case eDSTM:
      printf("eDSTM: READ( %s, %s )\n", p->name, p->qstr);
                break;
    default: fprintf(stderr, "******* An error in expressions!\n");
             break;
    }
  }
}

void print_stm ( tSTM* p ) {
  tEXP *te;
  tSTM *ts;

  if( p ) {
    switch( p->stm_id ) {
    case sMAIN:
```

```
  printf("sMAIN: ( ");
  print_exp( p->exp2 );
  printf(" ) ");
  print_stm( p->stm1 );
            break;
case sMDCL:
  printf("sMDCL: %s ( ", p->exp1->name );
  print_exp( p->exp2 );
  printf(" ) ");
  print_stm( p->stm1 );
            break;
case sBLOCK:
  printf("sBLOCK: {\n");
  print_stm( p->stm1 );
  printf(" }\n");
            break;
case sVDCL1:
  printf("sVDCL1: ");
  print_exp( p->exp1 );
  printf("%s\n", p->exp1->name);
            break;
case sVDCL2:
  printf("sVDCL2: ");
  print_exp( p->exp1 );
  print_stm( p->stm1 );
            break;
case sASTM:
  printf("sASTM: ");
              print_exp( p->exp1 );
            break;
case sRSTM:
  printf("sRSTM: return ");
              print_exp( p->exp1 );
            break;
case sISTM:
  printf("sISTM: if ( ");
  print_exp( p->exp1 );
  print_stm( p->stm1 );
  if (p->stm2) {
      printf("    else\n");
      print_stm( p->stm2 );
  }
            break;
```

```
    case sWSTM:
      printf("sWSTM: ");
      print_exp( p->exp1 );
                break;
    case sDSTM:
      printf("sDSTM: ");
      print_exp( p->exp1 );
                break;
    default: fprintf(stderr, "******* An error in statements!\n");
             break;
    }
    if (p->next) print_stm( p->next );
  }
}

void gen_exp ( tEXP* p ) {
  tEXP* te;
  if( p ) {
    switch( p->exp_id ) {
    case eTYPE:
      fprintf( yyout, "%s", c_types1[ p->ival - 1] );
                break;
    case eFORM:
      gen_exp( p->exp1 );
                  fprintf( yyout, " %s", p->name );
                  if (p->next) {
                    fprintf( yyout, ", " );
                    gen_exp( p->next );
                  }
                break;
    case eEXPR:
      gen_exp( p->exp1 );
                  gen_exp( p->next );
                break;
    case eADD:
      fprintf( yyout, " + " );
                  gen_exp( p->exp1 );
                  gen_exp( p->next );
                break;
    case eMINUS:
      fprintf( yyout, " - " );
                  gen_exp( p->exp1 );
                  print_exp( p->next );
```

```
            break;
case eMEXP: // Done.
  gen_exp( p->exp1 );
              gen_exp( p->next );
            break;
case eTIMES: // Done.
  fprintf( yyout, " * " );
              gen_exp( p->exp1 );
              gen_exp( p->next );
            break;
case eDIVIDE: // Done.
  fprintf( yyout, " / " );
              gen_exp( p->exp1 );
              print_exp( p->next );
            break;
case eINUM:
  fprintf( yyout, "%d", p->ival );
            break;
case eRNUM:
  fprintf( yyout, "%f", p->rval );
            break;
case eID:
  fprintf( yyout, "%s", p->name );
            break;
case ePAREN:  // Done.
  fprintf( yyout, " ( " );
  gen_exp( p->exp1 );
  fprintf( yyout, " ) " );
            break;
case eFUNC: // Done.
  fprintf( yyout, "%s", p->name );
  fprintf( yyout, " (" );
  gen_exp( p->exp1 );
  fprintf( yyout, ") " );
            break;
case eEQ:  // Done.
  gen_exp( p->exp1 );
  fprintf( yyout, " = " );
  gen_exp( p->next );
            break;
case eNE:  // Done.
  gen_exp( p->exp1 );
  fprintf( yyout, " != " );
```

```
  gen_exp( p->next );
            break;
case eGT:  // Done.
  gen_exp( p->exp1 );
  fprintf( yyout, " > " );
  gen_exp( p->next );
            break;
case eLT:  // Done.
  gen_exp( p->exp1 );
  fprintf( yyout, " < " );
  gen_exp( p->next );
            break;
case eGE:  // Done.
  gen_exp( p->exp1 );
  fprintf( yyout, " >= " );
  gen_exp( p->next );
            break;
case eLE:  // Done.
  gen_exp( p->exp1 );
  fprintf( yyout, " <= " );
  gen_exp( p->next );
            break;
case eAPARM: // Done.
  gen_exp( p->exp1 );
              print_exp( p->next );
  if (p->next) {
                fprintf( yyout, ", " );
                gen_exp( p->next );
              }
            break;
case eASSIGN1:
  fprintf( yyout, "%s = ", p->name );
              gen_exp( p->exp1 );
              fprintf( yyout, ";\n" );
            break;
case eASSIGN2:
  fprintf( yyout, "%s = %s;\n", p->name, p->qstr );
            break;
case eWSTM: // Done.
  int tmpwstm = ana_exptype( p );
  if (tmpwstm == 1) {
    fprintf( yyout, "tiny_writeint(");
    gen_exp( p->exp1 );
```

```
        fprintf( yyout, ", %s);\n", p->qstr );
      }
      if (tmpwstm == 2) {
        fprintf( yyout, "tiny_writereal(");
        gen_exp( p->exp1 );
        fprintf( yyout, ", %s);\n", p->qstr );
      }
      if (tmpwstm == 3) {
        fprintf( yyout, "tiny_writestr(");
        gen_exp( p->exp1 );
        fprintf( yyout, ", &%s);\n", p->qstr );
      }
              break;
    case eDSTM: // Done.
      int tmpdstm = ana_exptype( p );
      if (tmpdstm == 1) {
        fprintf( yyout, "tiny_readint(");
        fprintf( yyout, "&%s", p->name );
        fprintf( yyout, ", %s);\n", p->qstr );
      }
      if (tmpdstm == 2) {
        fprintf( yyout, "tiny_readreal(");
        fprintf( yyout, "&%s", p->name );
        fprintf( yyout, ", %s);\n", p->qstr );
      }
      if (tmpdstm == 3) {
        fprintf( yyout, "tiny_readstr(");
        fprintf( yyout, "%s", p->name );
        fprintf( yyout, ", %s);\n", p->qstr );
      }
              break;
    default: fprintf(stderr, "******* An error in expressions!\n");
             break;
    }
  }
}

void gen_code ( tSTM* p ) {
  tEXP *te;
  tSTM *ts;
  int t;

  if( p ) {
```

```
switch( p->stm_id ) {
case sMAIN:
  gen_exp( p->exp1->exp1 );
             fprintf( yyout, " main ( " );
             gen_exp( p->exp2 );
             fprintf( yyout, ")\n" );
             gen_code( p->stm1 );
           break;
case sMDCL:
  gen_exp( p->exp1->exp1 );
  fprintf( yyout, " " );
  gen_exp( p->exp1 );
  fprintf( yyout, " ( " );
  gen_exp( p->exp2 );
  fprintf( yyout, " )\n" );
  gen_code( p->stm1 );
           break;
case sBLOCK:
  fprintf( yyout, "{ ");
             gen_code( p->stm1 );
             fprintf( yyout, "}\n");
           break;
case sVDCL1:
  gen_exp( p->exp1 );
             fprintf( yyout, " %s;\n", p->exp1->name );
           break;
case sVDCL2:
  gen_exp( p->exp1 );
             fprintf( yyout, " " );
             gen_code( p->stm1 );
           break;
case sASTM:
  gen_exp( p->exp1 );
           break;
case sRSTM: // Done.
  fprintf( yyout, "return ");
  gen_exp( p->exp1 );
  fprintf( yyout, ";\n");
           break;
case sISTM: // Done.
  fprintf( yyout, "if ( ");
  gen_exp( p->exp1 );
  fprintf( yyout, " ) ");
```

```
      gen_code( p->stm1 );
      if (p->stm2) {
          fprintf(yyout, "  else\n");
          gen_code( p->stm2 );
      }
                break;
    case sWSTM: // Done.
      gen_exp( p->exp1 );
                break;
    case sDSTM: // Done.
      gen_exp( p->exp1 );
                break;
    default:    fprintf(stderr, "******* An error in statements!\n");
                break;
    }
    if (p->next) gen_code( p->next );
  }
}

int lookup( symNODE* p, char* s ) {
  int tmp = 0;
  if (p) {
    if (strcmp( p->name, s ) == 0) {
      tmp = p->type;
    } else {
      tmp = lookup( p->next, s );
    }
  }
  return tmp;
}

int ana_exptype( tEXP *p ) {
  // tINT = 1, tREAL = 2, tSTR = 3.
  int tmp = tINT;
  if (p) {
    //trying lookup first, if 0, checking for further expression
    int trylookup = lookup (symtab, p->name);
    if (trylookup != 0)
    {
      tmp = trylookup;
    }

    while (p->exp1) {
```

```
      if (p->next != NULL) //if the expression has next, chenking if real value
      {
        tEXP *nextP = p;
        while(nextP->next != NULL)
        {
          nextP = nextP->next;
          if (nextP->exp1->exp_id == 10)
          {
            tmp = tREAL; //real value used, expression is real
            break;
          }
        }
      }
      p=p->exp1;
    }
    return tmp;
  }
}

void init_all () {
    int i, j;
    // initialize gen_rw
    for (i=0; i<2; i++)
            for (j=0; j<3; j++) gen_rw[i][j] = 0;
    printf("mytiny2c: init_all() done!\n");
}

void gen_rwcode () {
    int i, j;

// Combine init_code
  fprintf( yyout, "#include <stdio.h>\n");
  fprintf( yyout, "#include <stdlib.h>\n");
  fprintf( yyout, "#include <string.h>\n");

// Real gen_rwcode
    if (gen_rw[0][0] == 1) {
          fprintf( yyout, "void tiny_readint ( int *x, char *s ) {\n");
        fprintf( yyout, "  printf(\"%%s \", s);\n");
        fprintf( yyout, "  scanf(\"%%d\", x);\n");
        fprintf( yyout, "}\n");
    }
    if (gen_rw[0][1] == 1) {
```

```
            fprintf( yyout, "void tiny_readreal ( float *x, char *s ) {\n");
        fprintf( yyout, "  printf(\"%%s \", s);\n");
        fprintf( yyout, "  scanf(\"%%f\", x);\n");
        fprintf( yyout, "}\n");
    }
    if (gen_rw[0][2] == 1) {
            fprintf( yyout, "void tiny_readstr ( char *x, char *s ) {\n");
        fprintf( yyout, "  printf(\"%%s \", s);\n");
        fprintf( yyout, "  scanf(\"%%s\", x);\n");
        fprintf( yyout, "}\n");
    }
    if (gen_rw[1][0] == 1) {
            fprintf( yyout, "void tiny_writeint ( int x, char *s ) {\n");
        fprintf( yyout, "  printf(\"%%s \", s);\n");
        fprintf( yyout, "  printf(\"%%d\\n\", x);\n");
        fprintf( yyout, "}\n");
    }
    if (gen_rw[1][1] == 1) {
            fprintf( yyout, "void tiny_writereal ( float x, char *s ) {\n");
        fprintf( yyout, "  printf(\"%%s \", s);\n");
        fprintf( yyout, "  printf(\"%%f\\n\", x);\n");
        fprintf( yyout, "}\n");
    }
    if (gen_rw[1][2] == 1) {
            fprintf( yyout, "void tiny_writestr ( char* x, char *s ) {\n");
        fprintf( yyout, "  printf(\"%%s \", s);\n");
        fprintf( yyout, "  printf(\"%%s\\n\", x);\n");
        fprintf( yyout, "}\n");
    }
}
```

4

TEST RUNS

4.1 TERMINAL OUTPUT OF "TEST.T"

```
mytiny2c: init_all() done!
MyTiny parse: type ok!
MyTiny parse: type ok!
MyTiny parse: type ok!
MyTiny parse: type ok!
MyTiny parse: block ok!
MyTiny parse: type ok!
MyTiny parse: type ok!
MyTiny parse: type ok!
MyTiny parse: type ok!
MyTiny parse: block ok!
mytiny2c: Parsing succeeded!
mytiny2c: starting gen_code!
eEXPR: eMEXP: eID: y
(end of eMEXP)
(end of eEXPR)
eEXPR: eMEXP: eID: x
(end of eMEXP)
(end of eEXPR)
```

4.2 C GENERATION OF "TEST.T"

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void tiny_readint ( int *x, char *s ) {
  printf("%s ", s);
  scanf("%d", x);
}
void tiny_writeint ( int x, char *s ) {
```

```
  printf("%s ", s);
  printf("%d\n", x);
}
int f2 ( int x, int y )
{ int z;
z = x * x - y * y;
return z;
}
int main ( )
{ int x;
tiny_readint(&x, "A41.input");
int y;
tiny_readint(&y, "A42.input");
int z;
z = f2 (x, y)  + f2 (y, x) ;
tiny_writeint(z, "A4.output");
}
```

4.3 C GENERATION OF "TEST2.T"

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void tiny_readreal ( float *x, char *s ) {
  printf("%s ", s);
  scanf("%f", x);
}
void tiny_writereal ( float x, char *s ) {
  printf("%s ", s);
  printf("%f\n", x);
}
int main ( )
{ float r;
tiny_readreal(&r, "Please input a real number as the radius of a circle:");
tiny_writereal(2 * 3.140000 * r, "The circumference of the circle is");
}
```

5

DISCUSSION

This programming assignment was more challenging, than the first one, it required even more theoretical application of compiler theory to practice, needed more time and effort to understand what needs to be done. The expression type analyzer was the most challenging of them all, really needed to understand what gives an expression its type, what is considered to be a real expression.

One of big challenges at the beginning was methods of debugging, until I figured out that t2c_tree.c can easily provide an output to the terminal, that is how I tracked how most of the variables behaved and what was the proper functionality of each part of expression, statement.

Overall it was an interesting challenge which I am happy I went through, proved me that the theoretical part of compilers and the practical has a very distinct difference and piqued my interest into focusing more on the lower level of programming than fully going to the high-level languages. There is much to be done in this field and it is interesting to see it develop, but most importantly, find a way to contribute to the clear practical improvements of the compiler development.