语义分析、中间代码生成程序实验：

18340146 计算机科学与技术 宋渝杰

实验题目

实验目的：构造 TINY + 的语义分析程序并生成中间代码

实验内容：构造符号表，用C语言扩展TINY的语义分析程序；构造 TINY + 的语义分析器；构造 TINY + 的中间代码生成器；

实验要求：能检查一定的语义错误，将 TINY + 程序转换成三地址中间代码。

实验过程

TINY+ 语言：

我的 TINY+ 语言和上一个词法语法实验完全一样，现在此再介绍一遍具体语言设计：

- 关键字：**IF ELSE WRITE READ RETURN BEGIN END MAIN INT REAL WHILE FOR**
- 分割符：`;` `,` `(` `)`
- 一元运算符：`+` `-` `*` `/` `%` (mod), `^` (xor)
- 多元运算符：`:=` `==` `!=` `>` `<`
- 标识符：标识符包含一个字母，后跟任意数量的字母或数字。 以下是标识符的示例：x，x2，xx2，x2x，End，END2。 请注意，**End** 是标识符，而 **END** 是关键字。 以下不是标识符：
  - `IF, WRITE, READ, ...`（关键字不算作标识符）
  - `2x`（标识符不能以数字开头）
  - 注释中的字符串不是标识符
- 数字：是一个数字序列，或一个数字序列，后跟一个小数点，再跟一个数字序列

```
Number -> Digits | Digits '.' Digits
Digits -> Digit | Digit Digits
Digit  -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

- 注释： `/**` 和 `**/` 之间的字符串，注释可以超过一行。

关于它的完全定义可以参考官网 https://jlu.myweb.cs.uwindsor.ca/214/language.htm

EBNF 文法

高级程序结构：

```
Program -> MethodDecl MethodDecl*
MethodDecl -> Type [MAIN] Id '(' FormalParams ')' Block
FormalParams -> [FormalParam ( ',' FormalParam )* ]
FormalParam -> Type Id

Type -> INT | REAL
```

声明：

```
Block -> BEGIN Statement* END

Statement -> Block
           | LocalVarDecl
           | AssignStmt
           | ReturnStmt
           | IfStmt
           | WriteStmt
           | ReadStmt
           | WhileStmt
           | ForStmt

LocalVarDecl -> INT Id ';'
              | REAL Id ';'

AssignStmt -> Id := Expression ';'
ReturnStmt -> RETURN Expression ';'
IfStmt -> IF '(' BoolExpression ')' Statement
        | IF '(' BoolExpression ')' Statement ELSE Statement
WriteStmt -> WRITE '(' Expression ',' QString ')' ';'
ReadStmt -> READ '(' Id ',' QString ')' ';'
WhileStmt -> WHILE '(' BoolExpression ')' Statement
ForStmt -> For '(' AssignStmt ';' BoolExpression ';' AssignStmt ')' Statement
QString is any sequence of characters except double quote itself, enclosed in
double quotes.
```

表达：

```
Expression -> AdditiveExpression  ( '^' AdditiveExpression )* // 位异或符号优先级最
低
AdditiveExpression -> MultiplicativeExpression  (( '+' | '-' )
MultiplicativeExpression)*
MultiplicativeExpression -> PrimaryExpr (( '*' | '/' ) PrimaryExpr)*
PrimaryExpr -> Num  // Integer or Real numbers
             | Id
             | '(' Expression ')'
             | Id '(' ActualParams ')'
BoolExpression -> Expression '==' Expression
                | Expression '!=' Expression
                | Expression '>' Expression
                | Expression '<' Expression
ActualParams -> [Expression ( ',' Expression)*]
```

构造符号表

符号表的构造要求如下：

- 函数名的作用范围为从函数声明的位置开始，到整个程序结束；
- 函数参数的作用范围为整个函数，函数 END 后参数不再起作用；
- 函数内部声明的变量的作用范围为从声明的位置开始，到函数 END 符号结束；
- for、while、if 语句内部声明的变量的作用范围为从声明的位置开始，到 for、while、if 的 END 符号结束；

实现过程：

我设计的算法主要是基于上一次语法实验所实现的语法树：因为对语法树进行一次前序遍历搜索就相当于对整个 tiny+ 文件代码进行了一次顺序遍历，而语法树的深度信息也表示了作用范围的深度，因此可以对语法树进行前序遍历处理，而**遍历到一个符号声明节点时，其子节点和右兄弟节点即为其作用范围。**

```
void printSymbolTable(struct TreeNode* root) { // 输出符号表
    deep++; // 深度信息
    while (root != NULL) {
        if (root->nodekind == MethodK) { // 函数声明节点
            switch (root->kind.method) {
                case MainK:
                    printSpaces();
                    printf("Main Method Symbol: %s\n", root->attr.name);
                    break;
                case NormalK:
                    printSpaces();
                    printf("Method Symbol: %s\n", root->attr.name);
                    break;
            }
        }
        ... // 函数参数、内部变量等同理
        for (int i = 0; i < 10; i++) printSymbolTable(root->child[i]); // 前序遍历
搜索
        root = root->next; // 右兄弟节点搜索
    }
    deep--; // 回溯
}
```

语义分析器

我所设计的语义规则如下：

- 对于表达式运算符 `+` `-` `*` `/` `%` (mod), `^` (xor) ，符号两端的操作数（或子表达式）必须是同一类型，而该表达式最终的类型也将被确定为该类型；
- 对于判断式运算符 `==` `!=` `>` `<` ，符号两端的操作数（或子表达式）必须是同一类型；
- 对于赋值语句运算符 `:=` ，符号左端的变量和右端的操作数（或子表达式）必须是同一类型；
- 在调用函数和变量时，函数名 / 变量名必须要在之前声明过，且处于其作用范围内；
- 在声明函数和变量时，函数名 / 变量名不能在之前声明过；

实现过程

对于表达式运算符两端的操作数（或子表达式）必须是同一类型，我采用递归分析表达式的方式：

- 先得到运算符左端操作数（或子表达式）的类型
- 再得到运算符右端操作数（或子表达式）的类型
- 最后判断两端类型是否相等

对于判断式运算符两端的操作数（或子表达式）必须是同一类型，也可以采用同样的方式：

```
int getType(struct TreeNode* root) { // 获取类型
    ...
    else if (root->kind.expression == OpK) {
        if (getType(root->child[0]) && getType(root->child[1]) && // 判断两端是否同
一类型
            getType(root->child[0]) != getType(root->child[1])) return -1;
```

```
        else return getType(root->child[0]);
    }
    ...
}

void checkType(struct TreeNode* root) { // 检查类型匹配问题
        ...
        if (root->nodekind == ExpressionK) {
            switch (root->kind.expression) { // 表达式和判断式
                case OpK:
                    if (getType(root->child[0]) && getType(root->child[1]) &&
                        getType(root->child[0]) != getType(root->child[1]))
judge = -1;
                    break;
            }
        }
        ...
}
```

对于赋值语句符号左端的变量和右端的操作数（或子表达式）必须是同一类型，右端的操作数（或子表达式）和上文同理，而左端的变量名则需要参考符号表：我采用一个 char 二维数组和 int 数组，分别存储多个变量名和它们对应的类型（ INT 为 1，REAL 为 2），之后需要查询变量名对应的类型的时候，遍历该表找出对应的类型即可：

```
char symbolTypeVector[100][100]; // 表
int symbolType[100], symTypeLen = 0;
int judge = 1

int getType(struct TreeNode* root) { // 获取type
    if (root->kind.statement == AssignK) { // 赋值语句
        for (int i = 1; i <= symTypeLen; i++) {
            if (strcmp(symbolTypeVector[i], root->attr.name) == 0) return
symbolType[i];
        }
        return 0;
    }
    ...
}

void checkType(struct TreeNode* root) { // 检查类型匹配问题
        ...
        else if (root->nodekind == StatementK) {
            switch (root->kind.statement) {
                case AssignK:
                    if (root->child[0] != NULL) { // 判断左变量和右表达式类型是否相同
                        if (root->child[0]->kind.expression == IdK &&
getType(root) && getType(root->child[0]) && getType(root) != getType(root-
>child[0])) judge = -1;
                    }
                    break;
                case IntDeclareK: // INT 变量声明
                    strcpy(symbolTypeVector[++symTypeLen], root->attr.name); //
写表
                    symbolType[symTypeLen] = 1;
                    break;
                case RealDeclareK: // REAL 变量声明
```

```
                    strcpy(symbolTypeVector[++symTypeLen], root->attr.name); //
写表
                    symbolType[symTypeLen] = 2;
                    break;
            }
        }
        ...
}
```

对于函数名 / 变量名必须要在之前声明过，且处于其作用范围内，同样也可以通过查询符号表来实现：如果表中不存在该函数名 / 变量名，则产生错误：

```
char symbolVector[100][100], funcVector[100][100]; // 表
int symLen = 0, funcLen = 0;
int c = 0;

void checkSymbol(struct TreeNode* root) { // 检查变量名和函数名问题
        ...
        else if (root->nodekind == StatementK) {
            switch (root->kind.statement) {
                ...
                case AssignK: // 变量名赋值
                    c = 0;
                    for (int i = 1; i <= symLen; i++) // 查表是否以声明该变量
                        if (strcmp(symbolVector[i], root->attr.name) == 0) c =
1;
                    if (c == 0) judge = -2; // ID call error
                    break;
            }
        }
        else if (root->nodekind == ExpressionK) {
            switch (root->kind.expression) {
                case IdK: // 调用变量
                    c = 0;
                    for (int i = 1; i <= symLen; i++) // 查表是否以声明该变量
                        if (strcmp(symbolVector[i], root->attr.name) == 0) c =
1;
                    if (c == 0) judge = -2; // ID call error
                    break;
                case MethodCallK: // 调用函数
                    c = 0;
                    for (int i = 1; i <= funcLen; i++) // 查表是否以声明该函数
                        if (strcmp(funcVector[i], root->attr.name) == 0) c = 1;
                    if (c == 0) judge = -3; // func call error
                    break;
            }
        }
        ...
}
```

而对于在声明函数和变量时，函数名 / 变量名不能在之前声明过，同样可以通过查表获得：如果表中已存在该函数名 / 变量名，则产生错误：

```
void checkSymbol(struct TreeNode* root) { // 检查变量名和函数名问题
        ...
        if (root->nodekind == MethodK) {
```

```
            switch (root->kind.method) {
                case NormalK: // 函数声明
                    c = 1;
                    for (int i = 1; i <= symLen; i++)
                        if (strcmp(symbolVector[i], root->attr.name) == 0) c =
0;
                    if (c == 0) judge = -1; // ID repeat declare error
                    strcpy(funcVector[++funcLen], root->attr.name);
                    break;
            }
        }
        else if (root->nodekind == TypeK) {
            switch (root->kind.type) {
                case IntTypeK:
                case RealTypeK: // 参数变量声明
                    c = 1;
                    for (int i = 1; i <= symLen; i++)
                        if (strcmp(symbolVector[i], root->attr.name) == 0) c =
0;
                    if (c == 0) judge = -1; // ID repeat declare error
                    strcpy(symbolVector[++symLen], root->attr.name);
                    break;
            }
        }
        else if (root->nodekind == StatementK) {
            switch (root->kind.statement) {
                case IntDeclareK:
                case RealDeclareK: // 函数内部变量声明
                    c = 1;
                    for (int i = 1; i <= symLen; i++)
                        if (strcmp(symbolVector[i], root->attr.name) == 0) c =
0;
                    if (c == 0) judge = -1; // ID repeat declare error
                    strcpy(symbolVector[++symLen], root->attr.name);
                    break;
                    ...
}
```

三地址中间代码

首先需要规定三地址中间代码的格式：由于题目并没有给出具体的格式要求，因此下面的三地址代码格式由我个人设计：

函数声明和参数：

源码	中间代码
INT f1(INT x)	METHOD f1 BEGIN RETURN TYPE INT PARAM INT x ... END
INT MAIN f()	MAIN METHOD f BEGIN RETURN TYPE INT ... END

变量声明：

源码	中间代码
INT ans;	INT i
REAL hello2;	REAL hello2

赋值语句：

源码	中间代码
i := 0	i := 0.00
z := x*x - y/y;	t1 := x * x t2 := y / y z := t1 - t2
z := z ^ x % y;	t1 := x % y z := z ^ t1

函数调用：

源码	中间代码
z := f1(x)	ARG x CALL f1 TO t1 z := t1
z := f2(y,x)	ARG y ARG x CALL f2 TO t2 z := t2

返回语句：

源码	中间代码
RETURN ans;	RETURN ans

if / while / for 语句：

源码	中间代码
IF (x != y) ...	t1 := x != y IFZ t1 GOTO L3 z := z + x LABEL L3

WHILE (x > y)
BEGIN
...
END
	LABEL L4
	t1 := x > y
	IFZ t1 GOTO LABEL L5
	...
	GOTO LABEL L4
	LABEL L5

FOR (i := 0; i < x; i := i + 1)
BEGIN
...
END
	i := 0.00
	LABEL L1
	t1 := i < x
	IFZ t1 GOTO LABEL L2
	...
	i := i + 1.00
	GOTO LABEL L1
	LABEL L2

read / write 语句：

源码	中间代码
READ(x, "A41.input");	READ "A41.input" to x
WRITE (z, "A4.output");	WRITE "A4.output" from z

实现过程

一些语句的中间代码输出实际上是和输出语法树相差不大的，所以稍微修改一下即可。而下面一些语句和输出语法树差别比较大，需要特别修改处理：

函数声明和参数：在我们的设计中，我们需要先输出 “METHOD + 函数名”，再输出 “BEGIN”，下一个是函数返回类型和参数，之后是函数体，最后是 “END”。而在我们的语法树中，第一、二个子节点就是函数返回类型和参数，之后都是函数体内容。因此我们可以先输出 “METHOD + 函数名 + BEGIN”，再按顺序递归输出所有子节点的中间代码，最后输出 “END” 即可：

```
void printMiddleCode(struct TreeNode* root) {                    // 输出中间代码
        ...
        if (root->nodekind == MethodK) {
            switch (root->kind.method) {
                case MainK:
                    printf("MAIN METHOD %s\nBEGIN\n", root->attr.name);
                    for (int i = 0; i < 10; i++) printMiddleCode(root-
>child[i]);
                    printf("END\n\n");
                    break;
                case NormalK:
                    printf("METHOD %s\nBEGIN\n", root->attr.name);
```

```
                    for (int i = 0; i < 10; i++) printMiddleCode(root-
>child[i]);
                    printf("END\n\n");
                    break;
            }
        }
        ...
}
```

赋值语句：赋值语句实际上涉及表达式计算和分析，而其中最难的地方在于中间变量：由于表达式可能过长，三地址中间代码不能一次性计算，因此就需要中间变量来存储表达式计算的中间结果。而我的做法是：由于我的语法树实际上存储了运算符优先级（即运算顺序）信息，对语法树表达式部分做一次后序遍历即得到真实的运算顺序，因此我们可以在后序遍历的基础上来递归记录中间变量名并返回至上层：

```
char* cal(struct TreeNode* root) { // 自底向上计算表达式
    char* t = (char*)malloc(50 * sizeof(char)), *t0, *t1;
    struct TreeNode* p;
    switch (root->kind.expression) {
        case OpK: // 如果是表达式
            t0 = cal(root->child[0]); // 递归计算左右子表达式
            t1 = cal(root->child[1]);
            switch (root->attr.token) {
                case ADD:
                    printf("   t%d := %s + %s\n", ++tempCnt, t0, t1);
                    sprintf(t, "t%d", tempCnt);
                    return t;
                ... // 其余运算符同理
            }
            break;
        case ConstK: // 常数直接返回值
            sprintf(t, "%.2f", root->attr.val);
            return t;
        case IdK: // 变量直接返回变量名
            sprintf(t, "%s", root->attr.name);
            return t;
        case MethodCallK: // 函数调用返回函数语句的操作方式（解释见下文）
            p = root->child[0]->child[0];
            while (p != NULL) {
                printf("   ARG %s\n", p->attr.name);
                p = p->next;
            }
            printf("   CALL %s TO t%d\n", root->attr.name, ++tempCnt);
            sprintf(t, "t%d", tempCnt);
            return t;
    }
}

void dispose(struct TreeNode* root) { // 处理表达式
    tempCnt = 0;
    if (root->child[0] != NULL && root->child[0]->kind.expression == OpK) { // 如
果是表达式
        char* t0 = cal(root->child[0]->child[0]); // 计算左右子表达式
        char* t1 = cal(root->child[0]->child[1]);
        switch (root->child[0]->attr.token) {
            case ADD: printf("   %s := %s + %s\n", root->attr.name, t0, t1);
break;
```

```
            ... // 其它运算符同理
        }
    }
    else {
        char* temp = cal(root);
        printf("   %s := %s\n", root->attr.name, temp);
    }
}

void printMiddleCode(struct TreeNode* root) { // 输出中间代码
        ...
        else if (root->nodekind == StatementK) {
            struct TreeNode* rootCopy = root;
            switch (root->kind.statement) {
                ...
                case AssignK:
                    dispose(rootCopy); // 处理表达式
                    break;
                    ...
}
```

函数调用：函数调用需要先输出它的参数，再输出 ”CALL + 函数名 + TO + 返回地址“，因此在顺序上输出它的子节点在前，输出函数节点的名称再后：

```
char* cal(struct TreeNode* root) { // 自底向上计算表达式
        ...
        case MethodCallK: // 函数调用
            p = root->child[0]->child[0];
            while (p != NULL) { // 先输出所有参数信息
                printf("   ARG %s\n", p->attr.name);
                p = p->next;
            }
            printf("   CALL %s TO t%d\n", root->attr.name, ++tempCnt);
            sprintf(t, "t%d", tempCnt);
            return t;
        ...
}
```

if / while / for 语句：它们中间代码的顺序都需要特殊处理：

- if：先输出条件判断式，再输出句式 ”IFZ + 条件判断式返回值 + GOTO LABEL 标签名“，之后输出 if 内部语句，最后是 “LABEL + 标签名”。
- while：按顺序输出 “LABEL + 标签名1”、条件判断式、”IFZ + 条件判断式返回值 + GOTO LABEL 标签名2“、 while 内部语句、“GOTO LABEL 标签名1“、“LABEL + 标签名2”。
- for：按顺序输出初始赋值、 “LABEL + 标签名1”、条件判断式、”IFZ + 条件判断式返回值 + GOTO LABEL 标签名2“、 for 内部语句、for 更新语句、“GOTO LABEL 标签名1“、“LABEL + 标签名2”。

因此它们的中间代码都要按特定顺序输出：

```
void printMiddleCode(struct TreeNode* root) {                    // 输出中间代码
        ...
        else if (root->nodekind == StatementK) {
            struct TreeNode* rootCopy = root;
            switch (root->kind.statement) {
                case IfK: // if 语句
                    rootCopy = root->child[0];
```

```
                    dispose2(rootCopy);
                    printf("   IFZ t1 GOTO L%d\n", ++labelCnt);
                    for (int i = 1; i < 10; i++) printMiddleCode(root-
>child[i]);
                    printf("LABEL L%d\n", labelCnt);
                    break;
                case WhileK: // while 语句
                    printf("LABEL L%d\n", ++labelCnt);
                    rootCopy = root->child[0];
                    dispose2(rootCopy);
                    printf("   IFZ t1 GOTO LABEL L%d\n", ++labelCnt);
                    for (int i = 1; i < 10; i++) printMiddleCode(root-
>child[i]);
                    printf("   GOTO LABEL L%d\n", labelCnt-1);
                    printf("LABEL L%d\n", labelCnt);
                    break;
                case ForK: // for 语句
                    rootCopy = root->child[0];
                    dispose(rootCopy);
                    printf("LABEL L%d\n", ++labelCnt);
                    rootCopy = root->child[1];
                    dispose2(rootCopy);
                    printf("   IFZ t1 GOTO LABEL L%d\n", ++labelCnt);
                    for (int i = 3; i < 10; i++) printMiddleCode(root-
>child[i]);
                    rootCopy = root->child[2];
                    dispose(rootCopy);
                    printf("   GOTO LABEL L%d\n", labelCnt-1);
                    printf("LABEL L%d\n", labelCnt);
                    break;
                    ...
}
```

测试功能整合

在完成了输出符号表、语义检测、中间代码生成功能之后，将其整合为最终的函数如下：

```
void check(struct TreeNode* root) { // 总测试
    printf("\n-------------------------- Symbol Table -------------------------
\n");
    struct TreeNode* rootcopy = root;
    printSymbolTable(rootcopy); // 输出符号表
    printf("\n----------------------- Semantic analysis ----------------------
\n");
    rootcopy = root;
    checkSymbol(rootcopy);
    if (judge == 1) { // 语义检查变量声明和调用无误
        rootcopy = root;
        checkType(rootcopy);
        if (judge == 1) { // 语义检查类型对应无误
            printf("No error ^_^\n");
            printf("\n-------------------------- Middle Code ------------------
--------\n");
            rootcopy = root;
            printMiddleCode(rootcopy); // 输出中间代码
        }
        else printf("Type Error >_<\n");
```

```
    }
    else if (judge == -1) printf("ID repeat declare Error >_<\n");
    else if (judge == -2) printf("ID call Error >_<\n");
    else printf("Func call Error >_<\n");
}
```

实验结果

实验测试文件：简单的 tiny+ 文件如下：（tiny1.tiny）

```
/** this is a comment line in the sample program **/
 INT f2(INT x, INT y )
 BEGIN
    INT z;
    REAL a;
    z := x*x - 2*2;
    RETURN z;
 END
 INT MAIN f1()
 BEGIN
    INT x;
    READ(x, "A41.input");
    INT y;
    READ(y, "A42.input");
    INT z;
    z := f2(x, y) + f2(y, x);
    z := x + y;
    WRITE (z, "A4.output");
 END
```

复杂的 tiny+ 文件代码如下：（tiny2.tiny）

```
/** this is another sample program **/
 INT f1(INT x)
 BEGIN
    INT i := 0;
    INT ans;
    FOR (i := 0; i < x; i := i + 1)
    BEGIN
       INT hello;
       ans := ans + i;
    END
    REAL hello2;
    RETURN ans;
 END
 INT f2(INT x, INT y )
 BEGIN
    INT z;
    z := x*x - y/y;
    IF (x != y) z := z + x;
    RETURN z;
 END
 INT MAIN f()
 BEGIN
    INT x;
    READ(x, "A41.input");
```

```
    INT y;
    READ(y, "A42.input");
    INT z;
    z := f1(x) + f2(y,x);
    WHILE (x > y)
    BEGIN
       z := z ^ x % y;
       x := x - y;
    END
    WRITE (z, "A4.output");
END
```

输出符号表测试：对两个文件分别进行符号表输出，结果如下：

```
------------------------- Symbol Table -------------------------
  Method Symbol: f2
     Param INT Symbol: x
     Param INT Symbol: y
     Declare INT Symbol: z
     Declare REAL Symbol: a
  Main Method Symbol: f1
     Declare INT Symbol: x
     Declare INT Symbol: y
     Declare INT Symbol: z
```

```
------------------------- Symbol Table -------------------------
  Method Symbol: f1
     Param INT Symbol: x
     Declare INT Symbol: i
     Declare INT Symbol: ans
        Declare INT Symbol: hello
     Declare REAL Symbol: hello2
  Method Symbol: f2
     Param INT Symbol: x
     Param INT Symbol: y
     Declare INT Symbol: z
  Main Method Symbol: f
     Declare INT Symbol: x
     Declare INT Symbol: y
     Declare INT Symbol: z
```

在符号表中清晰地表述了函数名、参数变量、内部声明变量，而且由缩进的长度可以说明该变量的作用范围。

语义分析测试：对两个文件分别进行符号表输出，结果都是没有问题的：

```
----------------------- Semantic analysis -----------------------
No error ^_^
```

下面分别修改一些代码，使得其产生问题，并测试我们程序的找错能力：

重复声明变量：

```
    INT z;
    INT z;
```

```
----------------------- Semantic analysis -----------------------
ID repeat declare Error >_<
```

使用未声明变量：

```
    z := 1;
    INT z;
```

```
---------------------- Semantic analysis ------------------------
ID call Error >_<
```

赋值语句两边类型不对：

```
    INT z;
    REAL a;
    z := a;
```

```
---------------------- Semantic analysis ------------------------
Type Error >_<
```

表达式计算两边类型不对：

```
INT f2(INT x, INT y )
BEGIN
   INT z;
   REAL a;
   z := x*x - a*a;
   RETURN z;
END
```

```
---------------------- Semantic analysis ------------------------
Type Error >_<
```

因此可以看出我的程序可以识别出一定的语义错误。

输出三地址中间代码：测试的结果如下：

test1.tiny：

```
------------------------- Middle Code -----------------------------
METHOD f2
BEGIN
    RETURN TYPE INT
    PARAM INT x INT y
    INT z
    REAL a
    t1 := x * x
    t2 := 2.00 * 2.00
    z := t1 - t2
    RETURN z
END

MAIN METHOD f1
BEGIN
    RETURN TYPE INT
    INT x
    READ "A41.input" to x
    INT y
    READ "A42.input" to y
    INT z
    ARG x
    ARG y
    CALL f2 TO t1
    ARG y
    ARG x
    CALL f2 TO t2
    z := t1 + t2
    z := x + y
    WRITE "A4.output" from z
END
```

test2.tiny 由于输出的中间代码过长，放在 txt/output.txt 内。

实验心得

本次实验主要工作内容为实现符号表、语义分析、中间代码生成，而这三个步骤在算法原理上都可以通过对上一个语法分析的语法分析树进行遍历得到，因此上一个实验有为这一个实验做了一些铺垫，也为这一个实验节省了一些工作量。

本次语义分析实验总体来说难度不大，但是由于时间的关系，我的程序实现的功能难免没有那么全面，但是也基本覆盖到了许多部分。而且在测试的过程中各种结果也比较合理，所以总的来说还是收获满满。