

TINY+ 语义分析程序及中间代码生成实验

TINY+ 语义分析程序及中间代码生成实验

实验目的

实验内容

实验要求

实验环境

TINY+语言&实现思路

实验过程

实验测试结果展示

心得体会

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
(中间代码优化)

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，（并简单地优化了中间代码）

实验环境

- Linux(Ubuntu)
- g++
- make

TINY+语言&实现思路

1. 一个用TINY+语言编写的程序包括变量的声明和语句序列两个部分。变量声明部分可以为空，但一个TINY+程序至少要包含一条语句。
 - a) 所有的变量在使用之前必须声明，并且每个变量只能被声明一次。
 - b) 变量以及表达式的类型可以是整型int，布尔类型bool或者字符串类型char，必须对变量的使用和表达式进行类型检查。

2. 构建语义分析器主要包括

- 符号表的设计以及符号表的相应操作
 - 符号表保存信息：变量的地址信息、变量被访问的行号
 - 变量以其变量名ID，通过链式hash储存查找到相应数据。
- 语义分析程序本身的操作，包括符号表的构建以及类型检查。
 - 符号表的创建：
 - 前序遍历语法树：
当遇到定义的语法结点时候，将其变量新增到哈希表中，并存储其行号位置；遇到含有变量的非定义语句的结点时，查找该变量位于哈希表中的位置，将该行号储存到哈希相应表项的next位置。
 - 类型检查：
 - 后续遍历语法树，对每一个结点进行语义判断：
 - 1. if语句结点中，if-test(child[0])的类型必定为boolean，
 - 2. while语句结点中，while-test(child[0])的类型必定为boolean，
 - 3. op算术运算结点中，child[0]、child[1]必须为integer类型
 - 4. op逻辑运算结点中，child[0]、child[1]必须为boolean
 - 5. 赋值运算结点中，child[0]、child[1]的类型必须一致
 - 6.
 - 在后续遍历语法树中，查看变量是否非法使用：
 - 1. 当结点为ID结点时候，在哈希表中查看是否已经定义。
 - 2. 如果该结点为定义语句结点，则判断此变量是否重复定义。
 - 3. 如果该变量已经定义，则将变量的type（类型）赋于定义时候的类型
 - 4. 进入上一层遍历，当其类型不符合上一层所需类型时候提示错误。（即如在赋值语句中，该变量的类型与赋值语句另一个子节点的类型不一致，提示语义错误）。

3. 三地址中间代码生成

- 原tiny中间代码生成器的实现位于cgen.c和cgen.h中。其接口为codeGen(TreeNode* syntaxTree,char* codefile).
主要功能为：
 - 函数的开始，产生一些注释以及建立运行时环境的指令。
 - 对语法树调用cGen函数，对每一个语法树结点生成相应指令

- Tiny+在tiny的基础上进行修改。其中codeGen函数将保持不变，修改其cGen函数即可，详细修改部分见后面实验过程。

4. 三地址中间代码优化

- 删除没有goto到的label
- 删除goto和同一个label紧接着的语句块
-

5. 语义分析

- 语义错误检查

- SEMANTIC_COND_BOOL_ERROR 条件判断语句必须是BOOL类型
- SEMANTIC_UNDEFINED_IDENTIFIER 未定义的变量
- SEMANTIC_OPERATION_BETWEEN_DIFFERENT_TYPES 符号用于不同的数据类型之间
- SEMANTIC_TYPE_CANNOT_BE_OPERATED 该类型不能用于该运算符
- SEMANTIC_CANNOT_ASSIGN_DIFFERENT_TYPE 不能赋予不同类型的值
- SEMANTIC_ILLEGAL_CHARACTER 语义分析中遇到的非法字符
- SEMANTIC_MISSING_SEMICOLON 缺少末尾的分号
- SEMANTIC_MULTIPLE_DECLARATIONS 多次声明

实验过程

1. 语义分析程序

- 构建符号表

先在之前语法树的基础上前序遍历语法树构建符号表，我这里是直接添加在了之前的语法分析的parser.cpp之中，利用符号表类实现的插入，查询等

操作，可以很快实现构建。以下是关于构建符号表相关代码截图：

```
/** 符号表的各种操作**/
class SymTable
{
private:
    //std::vector<Sym *> sym_table;
    std::map<std::string, Sym *> sym_table;

public:
    explicit SymTable() = default;
    Sym *insertSym(const std::string &name)
    {
        Sym *sym = new Sym();
        sym_table.insert(std::make_pair(name, sym));
        return sym;
    }
    Sym *findSym(const std::string &name)
    {
        if (sym_table.find(name) != sym_table.end())
            return sym_table[name];
        else
            return nullptr;
    }
    Sym *delSym(const std::string &name)
    {
        sym_table.erase(name);
    }
    void print()
    {
        std::cout << "Variable Type ValType\n-----\n";
    }
};
```

以上是符号表的相关代码，其中就实现了插入查找以及打印等功能函数，方便之后展示使用。

```
void print()
{
    std::cout << "Variable Type ValType\n-----\n";
    for (auto &i : sym_table)
    {
        std::cout << i.first;
        std::string tmp = i.first;
        for (int j = 0; j < 10 - tmp.length(); ++j)
            std::cout << " ";
        if (i.second->objType == 0)
            std::cout << "Function";
        else if (i.second->objType == 1)
            std::cout << "Value ";
        else
            std::cout << "Const ";

        if (i.second->valType == 0)
            std::cout << "Int " << std::endl;
        else if (i.second->valType == 1)
            std::cout << "Bool" << std::endl;
        else
            std::cout << "Str " << std::endl;
    }
    std::cout<<std::endl;
}
```

而插值我是在parser.cpp中同时实现，并没有独立开来。以下为部分截图。

```

while (token.kind == TK_INT || token.kind == TK_BOOL || token.kind == TK_STRING)
{
    Token type = token; //保存类型的符号
    // nextToken(); //跳过类型声明
    do
    {
        nextToken(); //跳过类型声明
        Token id = token; //保存标识符的符号
        if (match(ID))
        {
            if (symTable.findSym(id.s_val)) //如果重复声明, 报错
                throw_syntax_error(SEMANTIC_MULTIPLE_DECLARATIONS, lineno);

            Sym *sym = symTable.insertSym(id.s_val);
        }
    }
}

```

- 语义分析检查

根据上面的实现思路，后序遍历语法树，对于不同的节点不同的语句都要进行分类，讨论其语义的正确性。下面展示部分相关代码截图：

```

TreeNode *assign_stmt(Token id_token)
{
    //ASSIGN不再需要匹配
    std::string id_name = id_token.s_val;
    Sym *id_sym = symTable.findSym(id_name);

    /**语义分析模块**/
    if (id_sym == nullptr)
    {
        throw_syntax_error(SEMANTIC_UNDEFINED_IDENTIFIER, lineno);
    }

    TreeNode *cond_exp = nullptr;
    match(TK_ASSIGN);
    cond_exp = logical_or_exp();

    /**语义分析模块**/
    if (id_sym->valType != cond_exp->valType)
    {
        throw_syntax_error(SEMANTIC_CANNOT_ASSIGN_DIFFERENT_TYPE, lineno);
    }
}

```

以上代码即对应于实现思路中的**赋值运算结点**中，*child[0]*、*child[1]*的类型必须一致以及**当结点为ID结点时候，在表中查看是否已经定义。**

```

TreeNode *while_stmt()
{
    //TK_WHILE不再需要匹配
    TreeNode *cond_exp = nullptr, *rep_stmt = nullptr;
    cond_exp = logical_or_exp();

    /**语义分析模块**/
    if (cond_exp->valType != VT_BOOL)
    {
        throw_syntax_error(SEMANTIC_COND_BOOL_ERROR, lineno);
    }

    match(TK_DO);
    rep_stmt = stmt_sequence();
    match(TK_END);

    //return TreeNode::create_Node(WHILE_STMT, cond_exp, rep_stmt);
    //todo
    return TreeNode::create_Node(WHILE_STMT, cond_exp, rep_stmt);
}

```

以上代码即对应于实现思路中的**while语句结点**中，*while-test(child[0])*的类型

必定为boolean，否则就会抛出错误。

```
TreeNode *logical_and_exp()
{
    TreeNode *log_and_exp = nullptr, *log_or_exp = nullptr;
    log_and_exp = comparison_exp();
    if (token.kind == TK_AND)
    {
        match(TK_AND);
        log_or_exp = logical_and_exp();
    }
    else
    {
        return log_and_exp; // 减少树的深度
    }
    TreeNode *ret_node = TreeNode::create_Node(LOG_AND_EXP, log_and_exp, log_or_exp);
    /** 语义分析模块***/

    //and之间两个表达式有一个不是bool类型，错误
    if (log_and_exp->valType != VT_BOOL || log_or_exp->valType != VT_BOOL)
    {
        throw_syntax_error(SEMANTIC_TYPE_CANNOT_BE_OPERATED, lineno);
    }
    ret_node->valType = VT_BOOL;
    return ret_node;
}
```

以上代码即对应于实现思路中的**op逻辑运算结点中**，**child[0]**、**child[1]**必须为**boolean**，否则就会抛出错误。

```
TreeNode *add_exp()
{
    TreeNode *mu_exp = nullptr, *ad_exp = nullptr;
    mu_exp = mul_exp();
    std::vector<TokenType> add_op_list{TK_ADD, TK_SUB};
    NodeType type;

    if (!ad_exp)
        return mu_exp;

    TreeNode *ret_node = TreeNode::create_Node(type, mu_exp, ad_exp);
    /** 语义分析模块***/
    if (ad_exp)
    {
        //乘号用在非整型时，报错
        if (mu_exp->valType != VT_INT || ad_exp->valType != VT_INT)
        {
            throw_syntax_error(SEMANTIC_TYPE_CANNOT_BE_OPERATED, lineno);
        }
    }
    ret_node->valType = VT_INT;
    return ret_node;
}
```

以上代码即对应于实现思路中的**op算术运算结点中**，**child[0]**、**child[1]**必须为**integer类型**，否则就会抛出错误。

对于不同的节点都要做不同的语义检查，此处限于篇幅就不一一展示，详见代码文件。

2. 生成中间代码

生成中间代码我是在generate.h和generate.cpp中实现的。利用之前的parser函数获取语法树根节点，然后对其进行后序遍历。其中cgen函数修改如下：

```
void cgen() {
    //midCodeVec = MidCodeVec();
    // analyzingStack = AnalyzingStack();
    has_error = false;
    TreeNode *root = parse();
    if (has_error) {
        fprintf(code, "Compile Error...\n");
        return;
    }
    traversal_back(root);
    if (is_optimized)
        midCodeVec.shrink_codes();

    symTable.print();
    printMidCodeVec();
}
```

其中遍历如下：

```
void traversal_back(TreeNode *root) {
    if (!root)
        return;
    TreeNode *c1 = root->child[0], *c2 = root->child[1], *c3 = root->child[2];
    // 来一遍后续遍历

    /**REPEAT 语句需要在执行语句前加LABEL**/
    /**REPEAT 语句需要在执行语句前加LABEL**/
    if (root->nodeType == REPEAT_STMT) {
        emit(C_LABEL, get_label_num_str());
        set_label_start_of(root, label_num);
    }
    traversal_back(c1);

    /**IF 语句需要在then, else 语句加LABEL**/
    /** 直接在父节点操作，如果深入到子节点，就没法确定是不是IF 语句的孩子了**/
    if (root->nodeType == IF_STMT || root->nodeType == WHILE_STMT) {
        emit(C_LABEL, get_label_num_str());
        set_label_start_of(c2, label_num);
    }
    traversal_back(c2);
    if (root->nodeType == IF_STMT) {
        if (c3) {
            // 如果else子句存在，需要在then子句后面加个goto，还要额外的回填操作
            // 否则else子句会紧接then子句后执行，这是不可能的
            int nextlist_index = emit(C_GOTO, "XXX");
            tfList.makelist(NEXTLIST, root, nextlist_index);
        }

        emit(C_LABEL, get_label_num_str());
        set_label_start_of_else_of(root, label_num);
    } else if (root->nodeType == REPEAT_STMT) {
        // REPEAT 语句需要在语句后再加个LABEL
        emit(C_LABEL, get_label_num_str());
        set_label_end_of(root, label_num);
    } else if (root->nodeType == WHILE_STMT) {
        // WHILE 语句需要再执行语句紧接一个GOTO，指向判断前面
        int nextlist_index = emit(C_GOTO, "XXX");
        tfList.makelist(NEXTLIST, c2, nextlist_index); // c2或者root都可以，因为next都指向判断

        // WHILE 语句需要在语句后再加个LABEL
        emit(C_LABEL, get_label_num_str());
        set_label_end_of(root, label_num);
    }
    traversal_back(c3);
    if (root->nodeType == IF_STMT) {
        if (c3) {
            emit(C_LABEL, get_label_num_str()); // else执行后加一个label
            tfList.backpatch(NEXTLIST, root, label_num); // then的next指向该label
        }
    }
}
```


其中对不同的语句应做不同的操作，如添加goto，回填等等，详见其中注释。

```

    // **** 中间代码生成——加减乘除****/
    /** 要从分析栈中弹出前两个元素***/
    /** 加减乘除的结果作为中间结果保存**/
    case ADD_EXP:
    {
        s2_str = analyzingStack.pop().val;
        s1_str = analyzingStack.pop().val;
        t_str = get_t_num_str();
        emit(C_ADD, t_str, s1_str, s2_str);
        analyzingStack.push(root, t_str);
        break;
    }
    case SUB_EXP:
    {
        s2_str = analyzingStack.pop().val;
        s1_str = analyzingStack.pop().val;
        t_str = get_t_num_str();
        emit(C_SUB, t_str, s1_str, s2_str);
        analyzingStack.push(root, t_str);
        break;
    }
    case DIV_EXP:
    {
        s2_str = analyzingStack.pop().val;
        s1_str = analyzingStack.pop().val;
        t_str = get_t_num_str();
        emit(C_DIV, t_str, s1_str, s2_str);
        analyzingStack.push(root, t_str);
        break;
    }
    case MUL_EXP:
    {
        s2_str = analyzingStack.pop().val;
        s1_str = analyzingStack.pop().val;
        t_str = get_t_num_str();
    }
}
```

此处限于篇幅，仅展示部分，详见generate.cpp文件。

而为了便于中间代码生成的实现我还定义MidCodeVec类来存放中间代码。那么生成完毕输出就十分简单遍历就行了。而且其中还构造了许多功能函数可用于删除不必要的中间代码进行优化。相关代码截图如下：

```

class MidCodeVec
{
public:
    MidCodeVec()
    {
        codeVec = std::vector<MidCode>();
    }

    std::vector<MidCode> codeVec;

    int push(const MidCode &code)
    {
        codeVec.push_back(code);
        return codeVec.size() - 1;
    }

    int push(CodeType type, const std::string &s1, const std::string &s2, const std::string &t)
    {
        codeVec.emplace_back(type, s1, s2, t);
        return codeVec.size() - 1;
    }

    /**
     * @brief 擦除中间代码集合中的某一行中间代码
     * @details 仅用于擦除多余的goto语句!
     */
    void remove(const int &index)
    {
    }
}
```

```

/**
 * @brief 获取某一行的中间代码字符串表现形式，前面有一个行号
 * */
std::string get_item_str(int index)
{
    std::string tmp = codeVec[index].to_str();
    std::stringstream ss;
    ss << (index + 1) << " " << tmp;
    return ss.str();
}

std::string num_to_str(int num)
{
    std::stringstream ss;
    ss << num;
    return ss.str();
}

/**
 * @brief 用于拉链回填，填goto的终点
 * @param index 要填哪个语句
 * @dest 终点处
 * */
void set_goto_destination(int index, int dest)
{
    codeVec[index].t = num_to_str(dest);
}

```

输出生成的中间代码

```

void printMidCodeVec() {

    //todo just for test
    for (int i = 0; i < midCodeVec.codeVec.size(); i++) {
        //std::cout << midCodeVec.get_item_str(i) << std::endl;
        fprintf(code, "%s\n", midCodeVec.get_item_str(i).c_str());
    }
}

```

3. 简单的代码优化

根据之前的实现思路，删除没有goto到的label，删除goto和同一个label紧接着的语句块，可以实现以下代码：

```
/**
 * @brief 简单的中间代码简化
 */
void shrink_codes()
{
    std::map<std::string, int> label_goto;
    // 删除没有被goto到的Label
    for (int i = 0; i < codeVec.size(); i++)
    {
        if (codeVec[i].type == C_GOTO || codeVec[i].type == C_IF)
        {
            label_goto[codeVec[i].t]++;
        }
    }

    // 要倒过来删!
    for (int i = codeVec.size() - 1; i >= 0; i--)
    {
        if (codeVec[i].type == C_LABEL)
        {
            if (label_goto[codeVec[i].t] == 0)
            {
                remove(i);
            }
        }
    }

    // 删除goto和同一个Label紧接的语句块
    for (int i = codeVec.size() - 1; i >= 1; i--)
    {
        if (codeVec[i].type == C_LABEL && codeVec[i - 1].type == C_GOTO &&
            codeVec[i].t == codeVec[i - 1].t && label_goto[codeVec[i].t] == 1)
        {
            remove(i);
            remove(i - 1);
            i--;
        }
    }
}
```

实验测试结果展示

- 输入make得到可执行程序main，并在命令行输入参数运行，测试Tiny源程序以及输出符号表及三地址中间代码如下图：

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# make
g++ -lpthread -lm -Iinclude -g -w src/generate.cpp src/main.cpp src/parser.cpp src/print.cpp src/scan.cpp -o bin/main
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny

TINY COMPILATION: test.tny
Variable  Type  ValType
-----
A          Value  Int
B          Value  Int
C          Value  Int
D          Value  Int
fact       Value  Int
x          Value  Int
y          Value  Bool
z          Value  Str

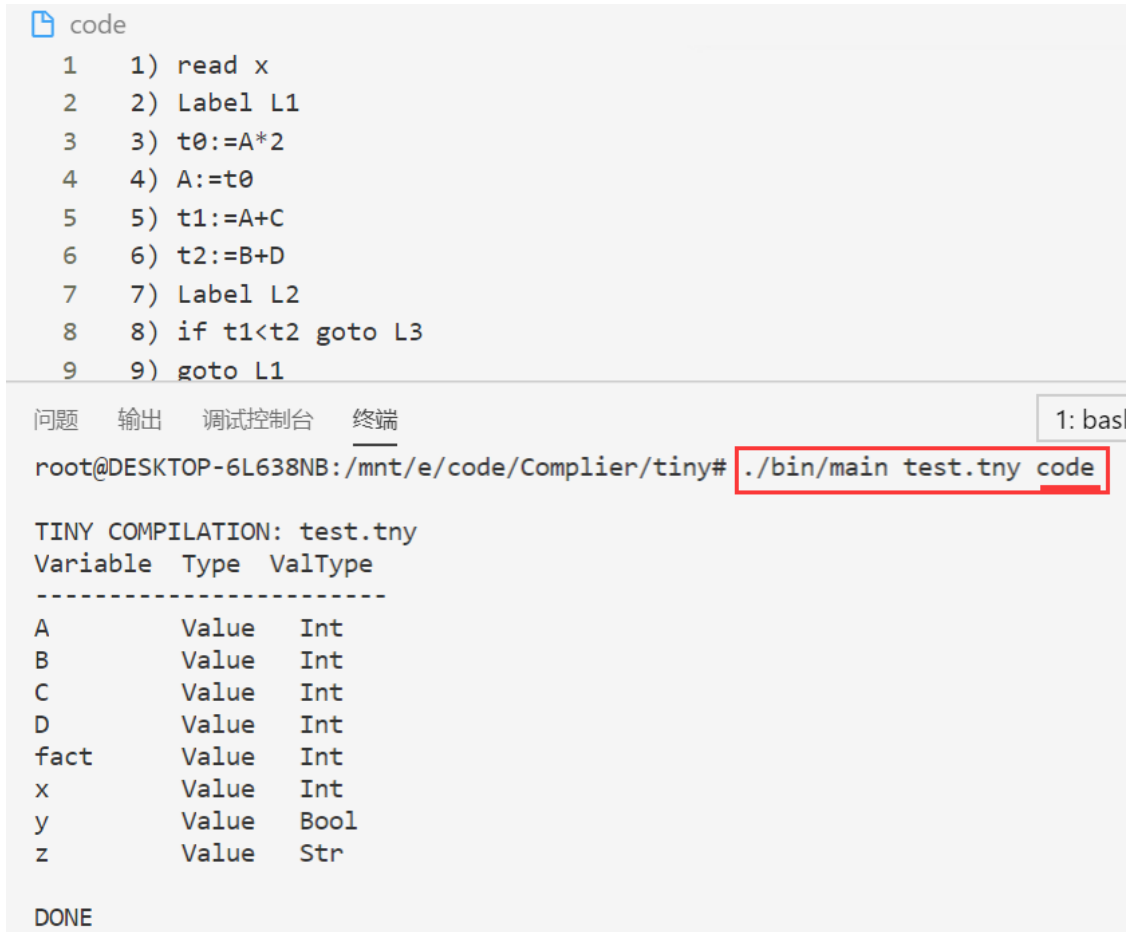
1) read x
2) Label L1
3) t0:=A*2
4) A:=t0
5) t1:=A+C
6) t2:=B+D
7) Label L2
8) if t1<t2 goto L3
9) goto L1
10) Label L3
11) t3:=B+C
12) t4:=A+t3
13) Label L4
14) if t4<10 goto L4
15) goto L6
16) Label L5
17) t5:=B+3
18) B:=t5
19) goto L4
20) Label L6
21) Label L7
22) if x<10 goto L8
23) goto L9
24) Label L8
25) if x>5 goto L10
26) goto L9
27) Label L9
28) if x<9 goto L10
29) goto L11
30) Label L10
31) fact:=4
32) goto L12
33) Label L11
34) fact:=6
35) Label L12
DONE
```

经检验，其代码跳转正确无误，符合代码逻辑。

- 其中测试的test.tny如下：

```
test.tny
1  int x,fact,A,B,C,D;
2  bool y;
3  string z;
4
5  {This is a comment.}
6  read x;
7
8  repeat
9      A:=A*2
10 until (A+C) < (B+D);
11
12 while (A+B+C) < 10 do
13     B := B + 3
14 end;
15
16 if x < 10 and x > 5 or x < 9 then
17     fact := 4
18 else
19     fact := 6
20 end;
21
```

- 输出到code文件



```
code
1  1) read x
2  2) Label L1
3  3) t0:=A*2
4  4) A:=t0
5  5) t1:=A+C
6  6) t2:=B+D
7  7) Label L2
8  8) if t1<t2 goto L3
9  9) goto L1
```

问题 输出 调试控制台 终端 1: basl

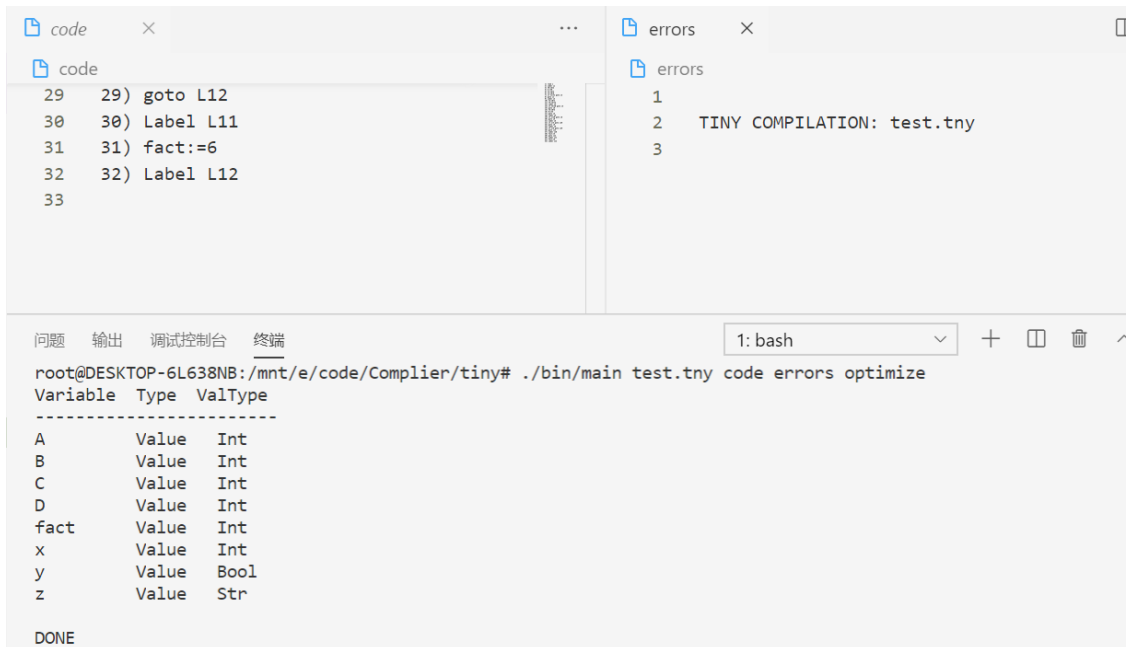
```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny code
```

TINY COMPILATION: test.tny

Variable	Type	ValType
A	Value	Int
B	Value	Int
C	Value	Int
D	Value	Int
fact	Value	Int
x	Value	Int
y	Value	Bool
z	Value	Str

DONE

- 输出错误报告信息到errors文件，输出优化过的代码



```
code errors
```

```
code
29 29) goto L12
30 30) Label L11
31 31) fact:=6
32 32) Label L12
33
```

```
errors
1
2 TINY COMPILATION: test.tny
3
```

问题 输出 调试控制台 终端 1: bash

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny code errors optimize
```

Variable	Type	ValType
A	Value	Int
B	Value	Int
C	Value	Int
D	Value	Int
fact	Value	Int
x	Value	Int
y	Value	Bool
z	Value	Str

DONE

可见生成的中间代码有35行优化去掉冗余成了32行，经验证逻辑仍然正确，符号表输出正确，errors中也并无报错。

- 测试语义错误

在errors.h中枚举了以下这些语义错误，下面将展示部分错误的报错。

```
* =====↓下面是语义错误=====
* SEMANTIC_COND_BOOL_ERROR 条件判断语句必须是BOOL类型
* SEMANTIC_UNDEFINED_IDENTIFIER 未定义的变量
* SEMANTIC_OPERATION_BETWEEN_DIFFERENT_TYPES 符号用于不同的数据类型之间
* SEMANTIC_TYPE_CANNOT_BE_OPERATED 该类型不能用于该运算符
* SEMANTIC_CANNOT_ASSIGN_DIFFERENT_TYPE 不能赋予不同类型的值
* SEMANTIC_ILLEGAL_CHARACTER 语义分析中遇到的非法字符
* SEMANTIC_MISSING_SEMICOLON 缺少末尾的分号
* SEMANTIC_MULTIPLE_DECLARATIONS 多次声明
* **/
```

○ 未定义变量

```
test.tny
1  int x,fact,A,B,C,D;
2  bool y;
3  string z;
4  k:=2
5  {This is a comment.}
6  read x;
7
8  ∨ repeat
9  |   A:=A*2
```

问题 输出 调试控制台 终端

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny

TINY COMPILATION: test.tny
An Error is detected at line 4: Undefined Identifier
```

○ 条件判断必须为Bool类型

```
test.tny
13  |   B := B + 3
14  end;
15
16  if 1
17  |   fact := 4
18  else
19  |   fact := 6
20  end;
21
```

问题 输出 调试控制台 终端

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny

TINY COMPILATION: test.tny
An Error is detected at line 17: The value of expression must be boolean
An Error is detected at line 17: An Syntax Error is found:
Compile Error...
DONE
```

- 符号用于不同数据类型之间

```
test.tny
1  int x,fact,A,B,C,D;
2  bool y;
3  string z;
4  y:=y*2
5  {This is a comment.}
6  read x;
7
8  repeat
9      A:=A*2
```

问题 输出 调试控制台 终端 1: bash

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny

TINY COMPILATION: test.tny
An Error is detected at line 4: This type cannot be operated by this operator.
Compile Error...
DONE
```

- 多次重复声明同一变量

```
test.tny
1  int x,fact,A,B,C,D;
2  bool y;
3  string z;
4  string z;
5  {This is a comment.}
6  read x;
7
8  repeat
9      A:=A*2
```

问题 输出 调试控制台 终端 1: bash

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny

TINY COMPILATION: test.tny
An Error is detected at line 4: Cannot declare the same identifier again!
Compile Error...
DONE
```

- 赋予不同类型的值

```
test.tny
1  int x,fact,A,B,C,D;
2  bool y;
3  string z;
4
5  {This is a comment.}
6  read x;
7  y:=1;
8  repeat
9      A:=A*2
```

问题 输出 调试控制台 终端 1:

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny

TINY COMPILATION: test.tny
An Error is detected at line 7: Cannot Assign different type
Compile Error...
DONE
```

- 本程序还能其他语义错误，此处就不一一展示，略占篇幅，欢迎尝试。

心得体会

语义分析程序和中间代码生成应该是这三个实验中较难的一个，好在前两个实验的代码较为有条理，为后面实验做了铺垫，实现起来才轻松许多。但是，我仍然需要阅读了许多的相关blog以及借鉴许多的代码才能完成，足以见得此次实验的难度，充分考验了我对这部分知识的理解与应用能力。经过这次实验，我对语义分析的实现以及三地址中间代码的生成有了更深的理解，并实现了从课堂理论知识到实际应用实现的映射，获益匪浅。