

《编译原理》课程实验报告

一、前言

对于此次的编译器设计，我们选择在《编译原理与实践》给予的 C- 框架基础上，构建我们的整个编译器。我们按照课本中的顺序，依次完成了前 1-7 章的各项内容，以课本提及的方法完成了词法分析，语法分析，语义分析，语法树的生成；并使用自己设计的方法，完成了mips汇编代码的生成，与语法树的可视化。

在此感谢南京大学许畅的编译原理课程实验报告，我们从中学到很多。[link](#)

代码的结构和用法可以在附录中查询。

组员分工

应承峻：词法分析、语法分析，语义分析，符号表，报告撰写

黄炯睿：生成中间代码，中间代码优化，生成mips汇编代码，语法树可视化，报告撰写

二、词法分析

1. 词法分析的目标

将文件的输入字符序列转换标记 (token) 序列的过程

2. 词法分析工具

通过Lex进行词法分析，在 `parse.l` 文件中设定词法后，由Lex工具自动生成词法分析程序。

3. C Minus语言关键字

```
1  "if"           { /* other codes... */ return IF; }
2  "else"         { /* other codes... */ return ELSE; }
3  "while"        { /* other codes... */ return WHILE; }
4  "int"          { /* other codes... */ return INT; }
5  "void"         { /* other codes... */ return VOID; }
6  "return"       { /* other codes... */ return RETURN; }
```

4. C Minus语言专用符号

```
1  "<="          { /* other codes... */ return LE; }
2  ">="          { /* other codes... */ return GE; }
3  "=="          { /* other codes... */ return EQ; }
4  "!="          { /* other codes... */ return NE; }
5  "<"           { /* other codes... */ return ('<'); }
6  ">"           { /* other codes... */ return ('>'); }
7  "="           { /* other codes... */ return ('='); }
8  ","           { /* other codes... */ return (',' ); }
9  ";"           { /* other codes... */ return (';'); }
10 "+"           { /* other codes... */ return ('+'); }
11 "-"           { /* other codes... */ return ('-'); }
12 "*"           { /* other codes... */ return ('*'); }
13 "/"           { /* other codes... */ return ('/'); }
14 "%"           { /* other codes... */ return ('%'); }
```

```

15 "("          { /* other codes... */ return '('); }
16 ")"          { /* other codes... */ return (')'); }
17 "["          { /* other codes... */ return ('['); }
18 "]"          { /* other codes... */ return (']'); }
19 "{"          { /* other codes... */ return ('{'); }
20 "}"          { /* other codes... */ return ('}'); }

```

5. C Minus语言标识符和其他标记

```

1 /*
2  * L          [a-zA-Z_]
3  * D          [0-9]
4  * LD         {L}|{D}
5  */
6
7 {L}{LD}*      { /* other codes... */ return IDENTIFIER;}
8 {D}+          { /* other codes... */ return CONSTANT; }
9 [\t\v\n\f]   { /* this is white space */ }

```

6. C Minus语言注释

```

1 "/*"          { }
2 "//"[^\n]*    { }

```

7. 词法分析为后续分析做的准备

在处理关键字时，先会通过调用 `col()` 来计算该关键字当前所属的列，便于后续定位错误，`col` 函数如下：

```

1 /*global part*/
2 int column = 0;
3 %option yylineno
4
5 void col() {
6     int i = 0;
7     while (i < strlen(yytext)) { //遍历该词法单元
8         char ch = yytext[i];
9         switch (ch) {
10             case '\n': //如果是换行符说明开启了新的一列
11                 column = 0;
12                 break;
13             case '\t': //如果是转义字符TAB，则需要按4个空格对齐
14                 column += 4 - (column % 4);
15                 break;
16             default: //逐字符递增
17                 column++;
18         }
19         i++;
20     }
21 }

```

在解析到记号后，会构造语法树节点，指明该节点的类型和值，以及该记号，如：

```

1  "+"      {
2      /* other codes... */
3      yy1val.syntax_tree = new SyntaxTree("addop", yytext);
4      yy1val.syntax_tree->lineNo = yylineno;
5      yy1val.syntax_tree->colNo = column;
6      return ('+');
7  }

```

最后，词法分析生成记号串，供语法分析进行分析，如：

```

1  "="      { /* other codes... */ return EQ; }

```

三、语法分析

1. 语法分析的目标

语法分析器根据从词法分析过程中获得的单词流，进行语法检查、并构建由输入的单词组成的语法树。

2. 语法分析工具

通过Yacc进行语法分析，在 `parse.y` 文件中设定语法后，由Yacc工具自动生成语法分析程序。

3. 语法树数据结构的定义

```

1  class SyntaxTree
2  {
3      int lineNo;
4      int colNo;
5      string type;
6      string value;
7      vector<SyntaxTree*> children;
8      NodeAttribute attr;
9  };
10 struct NodeAttribute
11 {
12     VariableType dataType = VARERROR;
13     union DataValue
14     {
15         int ivalue;
16     } dataValue;
17 };

```

语法树的每一个节点主要包含6个域，`lineNo` 和 `colNo` 表示当前节点所对应的行号和列号；`type` 表示该节点的词法或文法单元的类型，`value` 表示该词法单元的节点的值（如一个类型为 `identifier` 词法单元对应的一个可能的 `value` 是 `x`，即定义一个变量 `x`）；`children` 是一个 `vector` 类型的数组，它表示该语法树节点所有的子节点，使用 C++ 中的 `vector` 数据类型能够很方便地对一个节点的子节点进行增加、删除和遍历操作；`attr` 是一个自定义的 `NodeAttribute` 类型的变量，它表示该语法树节点中的属性（如值类型、值等），该字段在语义分析过程中用于属性计算。

4. C Minus语言的BNF文法

程序由声明的列表(或序列)组成，声明可以是函数或变量声明，顺序是任意的，至少必须有一个声明。

所有的变量和函数在使用前必须声明。

程序中最后的声明必须是一个函数声明，名字为 `main`。

```
1  program:
2      declaration-list {
3          $$ = new SyntaxTree("program");
4          $$->add($1);
5          root = $$;
6      }
7      ;
8
9  declaration-list:
10     declaration-list declaration {
11         $$ = new SyntaxTree("declaration-list");
12         $$->add($1)->add($2);
13     }
14     | declaration {
15         $$ = new SyntaxTree("declaration-list");
16         $$->add($1);
17     }
18     ;
19
20  declaration:
21     var-declaration {
22         $$ = $1;
23     }
24     | fun-declaration {
25         $$ = $1;
26     }
27     ;
```

变量声明或者声明了简单的整数类型变量，或者是基类型为整数的数组变量，索引范围从0到 `NUM-1`。

在 `C` 中仅有的基本类型是整型和空类型。在一个变量声明中，只能使用类型指示符 `int`，`void` 用于函数声明。

每个声明只能声明一个变量。

```
1  var-declaration:
2      type-specifier IDENTIFIER ';' {
3          $$ = new SyntaxTree("var-declaration");
4          $$->add($1)->add($2);
5      }
6      | type-specifier IDENTIFIER '[' CONSTINT ']' ';' {
7          $$ = new SyntaxTree("var-declaration");
8          $$->add($1)->add($2)->add($4);
9      }
10     ;
11
12  type-specifier:
13     INT {
14         $$ = $1;
15     }
16     | VOID {
17         $$ = $1;
18     }
19     ;
```

函数声明由返回类型指示符、标识符以及在圆括号内的用逗号分开的参数列表组成，后面跟着一个复合语句，是函数的代码。

如果函数的返回类型是 `void`，那么函数不返回任何值。

函数的参数可以是 `void`，或者一系列描述函数的参数。参数后面跟着方括号是数组参数，其大小是可变的。函数可以是递归的。

```
1 fun-declaration:
2     type-specifier IDENTIFIER '(' params ')' compound-stmt {
3         $$ = new SyntaxTree("fun-declaration");
4         $$->add($1)->add($2)->add($4)->add($6);
5     }
6     ;
7
8 params:
9     param-list {
10         $$ = new SyntaxTree("params");
11         $$->add($1);
12     }
13     | VOID {
14         $$ = new SyntaxTree("params");
15         $$->add($1);
16     }
17     ;
18
19 param-list:
20     param-list ',' param {
21         $$ = new SyntaxTree("param-list");
22         $$->add($1)->add($3);
23     }
24     | param {
25         $$ = new SyntaxTree("param-list");
26         $$->add($1);
27     }
28     ;
29
30 param:
31     type-specifier IDENTIFIER {
32         $$ = new SyntaxTree("param");
33         $$->add($1)->add($2);
34     }
35     | type-specifier IDENTIFIER '[' ']' {
36         $$ = new SyntaxTree("param");
37         $$->add($1)->add($2)->add(new SyntaxTree("[]"));
38     }
39     ;
```

复合语句由用花括号围起来的一组声明和语句组成并通过用给定的顺序执行语句序列。

局部声明的作用域等于复合语句的语句列表，并代替任何全局声明。

```
1 compound-stmt:
2     '{' local-declarations statement-list '}' {
3         $$ = new SyntaxTree("compound-stmt");
4         $$->add($2)->add($3);
5     }
```

```

6      ;
7
8  local-declarations:
9      {
10         $$ = new SyntaxTree("local-declarations");
11     }
12     | local-declarations var-declaration {
13         $$ = new SyntaxTree("local-declarations");
14         $$->add($1)->add($2);
15     }
16     ;
17
18  statement-list:
19      {
20         $$ = new SyntaxTree("statement-list");
21     }
22     | statement-list statement {
23         $$ = new SyntaxTree("statement-list");
24         $$->add($1)->add($2);
25     }
26     ;

```

表达式语句有一个可选的且后面跟着分号的表达式。这样的表达式通常求出它们一方的结果。

这类语句通常用于赋值和函数调用。

```

1  statement:
2      expression-stmt {
3          $$ = $1;
4      }
5      | compound-stmt {
6          $$ = $1;
7      }
8      | selection-stmt {
9          $$ = $1;
10     }
11     | iteration-stmt {
12         $$ = $1;
13     }
14     | return-stmt {
15         $$ = $1;
16     }
17     ;
18
19  expression-stmt:
20      expression ';' {
21         $$ = new SyntaxTree("expression-stmt");
22         $$->add($1);
23     }
24     | ';' {
25         $$ = new SyntaxTree("expression-stmt");
26     }
27     ;

```

if 语句通过表达式计算判断条件，true 值引起第一条语句的执行；false 值引起第二条语句的执行，如果它存在的话。

```

1 selection-stmt:
2     IF '(' expression ')' statement %prec IFX {
3         $$ = new SyntaxTree("selection-stmt");
4         $$->add($1)->add($3)->add($5);
5     }
6     | IF '(' expression ')' statement ELSE statement {
7         $$ = new SyntaxTree("selection-stmt");
8         $$->add($1)->add($3)->add($5)->add($7);
9     }
10    ;

```

`while` 语句用于重复执行表达式，并且如果表达式的求值为 `true` 则执行语句，当表达式的值为 `false` 时结束。

```

1 iteration-stmt:
2     WHILE '(' expression ')' statement {
3         $$ = new SyntaxTree("iteration-stmt");
4         $$->add($1)->add($3)->add($5);
5     }
6     ;

```

返回语句可以返回一个值也可无值返回，函数声明为 `void` 就没有返回值。

```

1 return-stmt:
2     RETURN ';' {
3         $$ = new SyntaxTree("return-stmt");
4         $$->add($1);
5     }
6     | RETURN expression ';' {
7         $$ = new SyntaxTree("return-stmt");
8         $$->add($1)->add($2);
9     }
10    ;

```

表达式是一个变量引用，后面跟着赋值符号 (等号) 和一个表达式，或者就是一个简单的表达式。

```

1 expression:
2     var '=' expression {
3         $$ = new SyntaxTree("expression");
4         $$->add($1)->add($3)->add($2);
5     }
6     | simple-expression {
7         $$ = new SyntaxTree("expression");
8         $$->add($1);
9     }
10    ;
11
12 var:
13     IDENTIFIER {
14         $$ = new SyntaxTree("var");
15         $$->add($1);
16     }
17     | IDENTIFIER '[' expression ']' {
18         $$ = new SyntaxTree("var");
19         $$->add($1)->add($3);

```

```
20     }
21     ;
```

简单表达式由无结合的关系操作符组成。

```
1  simple-expression:
2      additive-expression relop additive-expression {
3          $2->add($1)->add($3);
4          $$ = $2;
5      }
6      | additive-expression {
7          $$ = $1;
8      }
9      ;
10
11 relop:
12     LE { $$ = $1; }
13     | '<' { $$ = $1; }
14     | '>' { $$ = $1; }
15     | GE { $$ = $1; }
16     | EQ { $$ = $1; }
17     | NE { $$ = $1; }
18     ;
19
20 additive-expression:
21     additive-expression addop term {
22         $2->add($1)->add($3);
23         $$ = $2;
24     }
25     | term { $$ = $1; }
26     ;
27
28 addop:
29     '+' { $$ = $1; }
30     | '-' { $$ = $1; }
31     ;
32
33 term:
34     term mulop factor {
35         $2->add($1)->add($3);
36         $$ = $2;
37     }
38     | factor { $$ = $1; }
39     ;
40
41 mulop:
42     '*' { $$ = $1; }
43     | '/' { $$ = $1; }
44     | '%' { $$ = $1; }
45     ;
46
47 factor:
48     '(' expression ')' { $$ = $2; }
49     | var { $$ = $1; }
50     | call { $$ = $1; }
51     | CONSTANT { $$ = $1; }
52     ;
```


函数调用的组成是一个IDENTIFIER (函数名)，后面是用括号围起来的参数。

参数或者为空，或者由逗号分割的表达式列表组成，表示在一次调用期间分配的参数的值。

在调用之前必须声明，声明中参数的数目必须等于调用中参数的数目。

函数声明中的数组参数必须和一个表达式匹配，这个表达式由一个标识符组成表示一个数组变量。

```

1  call:
2      IDENTIFIER '(' args ')' {
3          $$ = new SyntaxTree("call");
4          $$->add($1)->add($3);
5      }
6      ;
7
8  args:
9      {
10         $$ = new SyntaxTree("args");
11     }
12     | arg-list {
13         $$ = new SyntaxTree("args");
14         $$->add($1);
15     }
16     ;
17
18  arg-list:
19     arg-list ',' expression {
20         $$ = new SyntaxTree("arg-list");
21         $$->add($1)->add($3);
22     }
23     | expression {
24         $$ = new SyntaxTree("arg-list");
25         $$->add($1);
26     }
27     ;

```

5. 语法分析为语义分析的铺垫

由于使用YACC使用 LALR(1) 进行自底向上的分析，因此经过语法分析后会生成一棵唯一的语法树，在语义分析阶段，将会对该语法树进行再一次遍历以检查不符合语义的节点。

四、语义分析

1. 语义分析的目标

语义分析是编译过程的一个逻辑阶段，其任务是对结构上正确的源程序进行上下文有关性质的审查和类型的检查，确保源程序没有语义上的错误。

2. 符号表的数据结构

符号表本质上是存储 Key-Value 键值对的哈希表，在C++中，我们可以使用STL标准模板库中的 map 容器来进行存储。在C++的底层实现中 map 是集合了数组、链表和红黑树的数据结构，它能够保持查找效率在 $O(\log N)$ 的同时尽可能减少容器所占用的空间。

在整个程序中，我们采用的方式是维护一个符号表栈(*Functional Style*)。假设当前函数 `f` 有一个符号表，表里有 `a`, `b`, `c` 这三个变量的定义。当编译器发现函数中出现了一个被 “{” 和 “}” 包含的语句块（即进入了一个新的作用域）时，它会将 `f` 的符号表压栈，然后新建一个符号表，这个符号表里只有变量 `a` 的定义。当语句块中出现任何表达式使用到某个变量时，编译器先查找当前的符号表，如果找到就使用这个符号表里的该变量，如果找不到则顺着符号表栈向下逐个符号表进行查找，使用第一个查找成功的符号表里的相应变量。如果查遍所有的符号表都找不到这个变量，则报告当前语句出现了变量未定义的错误。每当编译器离开某个语句块时，会先销毁当前的符号表，然后从栈中弹一个符号表出来作为当前的符号表。

根据上面的实现思路，我们定义新的数据类型：符号表(`Table`)，它本质就是由 `<string, Entry*>` 组成的键值对的集合。而 `SymbolTable` 类是符号表的集合，它包含了如下操作：

- 初始化符号表集合并往里面插入一张初始符号表： `SymbolTable()`
- 在符号表集合中查找某一标识符： `Entry* lookup(string name)`
- 在当前栈顶符号表中查找某一标识符： `Entry* find(string name)`
- 创建一张新的符号表并压栈： `void create()`
- 向栈顶符号表中插入某一标识符： `bool insert(Entry* e)`
- 弹出栈顶符号表： `bool remove()`
- 获取符号表集合中符号表的数量： `int deep()`
- 拷贝当前符号表中的参数到函数的作用域中： `void copyParams(Entry* e)`

```
1  typedef map<string, Entry*> Table;
2
3  class SymbolTable
4  {
5  public:
6
7      SymbolTable() {
8          create();
9      }
10
11     Entry* lookup(string name) {
12         for (int i = 0; i < tables.size(); i++) {
13             Table &t = tables[i];
14             Table::iterator iter = t.find(name);
15             if (iter != t.end()) {
16                 return iter->second;
17             }
18         }
19         return NULL;
20     }
21
22     Entry* find(string name) { //find symbol at current table
23         Table &t = tables[0];
24         Table::iterator iter = t.find(name);
25         if (iter != t.end()) {
26             return iter->second;
27         }
28         return NULL;
29     }
30
31     void create() {
32         tables.insert(tables.begin(), Table());
33     }
34 }
```

```

35     bool insert(Entry* e) {
36         if (e->varType == Void && e->symType != Function) {
37             return false;
38         }
39         Table &t = tables[0];
40         Table::iterator iter = t.find(e->identifier);
41         if (iter == t.end()) {
42             t[e->identifier] = e;
43             return true;
44         } else {
45             return false;
46         }
47     }
48
49     bool remove() {
50         if (tables.size() > 1) {
51             Table &t = tables[0];
52             for (Table::iterator iter = t.begin(); iter != t.end(); iter++)
53             {
54                 delete iter->second;
55             }
56             tables.erase(tables.begin());
57             return true;
58         }
59         return false;
60     }
61
62     int deep() {
63         return tables.size();
64     }
65
66     void copyParams(Entry* e) {
67         Table &t = tables[0];
68         for (Table::iterator iter = t.begin(); iter != t.end(); iter++) {
69             Entry* cp = new Entry(iter->second->symType,
70                                   iter->second->varType,
71                                   iter->second->identifier,
72                                   iter->second->num);
73             e->insertParam(cp);
74         }
75     }
76 private:
77     vector<Table> tables;
78 };

```

对于单个符号表中的每一个项 `Entry`，这个项应该包含符号的语义类型（变量、数组、参数、函数），符号的变量类型（VOID, INT, BOOL），符号名称；以及可选的项目：①如果是数组，则数组的大小，②如果是函数，则函数的参数信息将会被链接到这里用于后续检查函数调用时的参数列表。

```

1  enum SymbolType
2  {
3      Variable, Array, Parameter, Function, SEMERROR
4  };
5  enum variableType
6  {
7      Void, Boolean, Integer, VARERROR

```

```

8   };
9
10  class Entry
11  {
12  public:
13      Entry(SymbolType symType, VariableType varType, string identifier) {
14          this->symType = symType;
15          this->varType = varType;
16          this->identifier = identifier;
17          this->num = 0;
18      }
19      Entry(SymbolType symType, VariableType varType, string identifier, int
num) {
20          this->symType = symType;
21          this->varType = varType;
22          this->identifier = identifier;
23          this->num = num;
24      }
25      void insertParam(Entry* e) {
26          this->params.push_back(e);
27      }
28      SymbolType symType; //symbol type
29      VariableType varType; //variable type, and if symbol type is
Function, varType means return type
30      string identifier; //identifier name
31      int num; //array size
32      vector<Entry*> params; //if symbol type is Function, then all params
will store here
33  private:
34      Entry() {}
35  };

```

3. 语义分析的驱动程序

语义分析需要对语法树进行自底向上的分析，因此整个程序需要采取递归调用的形式，并且子节点递归调用的顺序也是从左到右。每当驱动程序遇到一个语法树节点时，会去判断该节点的类型，然后再调用对应的语义分析程序。

```

1  bool parse(SyntaxTree* root) {
2      if (root == NULL) return true;
3      if (root->type == "program") {
4          return parse(root->children[0]);
5      } else if (root->type == "declaration-list") {
6          return Declaration_List(root);
7      } else if (root->type == "var-declaration") {
8          return Var_Declaration(root);
9      } else if (root->type == "fun-declaration") {
10         return Fun_Declaration(root);
11     } else if (root->type == "params") {
12         return Params(root);
13     } else if (root->type == "param-list") {
14         return Param_List(root);
15     } else if (root->type == "param") {
16         return Param(root);
17     } else if (root->type == "compound-stmt") {
18         return Compound_Stmt(root);
19     } else if (root->type == "local-declarations") {

```

```

20     return Local_Declarations(root);
21 } else if (root->type == "statement-list") {
22     return Statement_List(root);
23 } else if (root->type == "expression-stmt") {
24     return Expression_Stmt(root);
25 } else if (root->type == "compound-stmt") {
26     return Compound_Stmt(root);
27 } else if (root->type == "selection-stmt") {
28     return Selection_Stmt(root);
29 } else if (root->type == "iteration-stmt") {
30     return Iteration_Stmt(root);
31 } else if (root->type == "return-stmt") {
32     return Return_Stmt(root);
33 } else if (root->type == "expression") {
34     return Expression(root);
35 } else if (root->type == "var") {
36     return Var(root);
37 } else if (root->type == "relation-operator") {
38     return Relop(root);
39 } else if (root->type == "addop") {
40     return Addop(root);
41 } else if (root->type == "mulop") {
42     return Mulop(root);
43 } else if (root->type == "call") {
44     return Call(root);
45 } else if (root->type == "identifier") {
46     return Identifier(root);
47 } else if (root->type == "integer") {
48     return INTEger(root);
49 }
50 return true;
51 }

```

具体的语义分析也采用递归的方式进行分析，如下述代码中，当碰到 `declaration-list` 类型的节点时，驱动程序会调用 `Declaration_List` 这一函数，然后这一函数又会去调用其每一个子节点的驱动程序，待所有子节点都分析完毕后，再返回。

```

1 bool Declaration_List(SyntaxTree* root) {
2     bool flag = true;
3     for (SyntaxTree* child: root->children) {
4         flag = flag && parse(child);
5     }
6     root->attr.dataType = VARERROR;
7     return flag;
8 }

```

4. 属性计算和语义验证

为了确保程序是符合内容的，需要对程序进行语义验证。在语义验证中，我们主要验证的是值类型 `VarType`，该属性是综合属性，因此我们可以直接在自底向上遍历语法树的同时完成属性计算和语义验证。

下面将会举3个具有代表性的例子来介绍属性计算和语义验证的过程：

例1：以 `Var-Declaration` 为例介绍一般类型的变量查找

当处理 `var-declaration` 类型的节点时，会先检查其第1个孩子的值（也就是这个变量声明的类型），如果不是要求的类型（`int`），则程序报错“变量类型不合法”。然后编译器会调用 `find` 函数在**栈顶符号表**中查找该变量（不遍历所有符号表的原因是变量可以在内部作用域定义，在内部作用域中将会隐藏外部作用域的相同变量），如果符号表中找到这一个符号，则报错“该变量已经被定义”，如果没有找到则将该符号插入到符号表中，并返回成功的结果。

```
1  bool Var_Declaration(SyntaxTree* root) {
2      bool flag = true;
3      if (root->children[0]->value == "int") {
4          Entry* e = stbl.find(root->children[1]->value);
5          if (e == NULL) {
6              if (root->children.size() == 2) { //normal variable
7                  e = new Entry(Variable, Integer, root->children[1]->value);
8              } else if (root->children.size() == 3) { //array
9                  int asize = atoi(root->children[2]->value.c_str());
10                 e = new Entry(Array, Integer, root->children[1]->value,
11                             asize);
12                 stbl.insert(e);
13                 flag = true;
14             } else {
15                 SeUtil::serror(root->children[1],
16                             "variable " + root->children[1]->value + "
17                 already exist.");
18                 flag = false;
19             }
20         } else {
21             SeUtil::serror(root->children[0],
22                             "invalid variable type " + root->children[0]-
23             >value);
24             flag = false;
25         }
26         root->attr.dataType = VARERROR;
27         return flag;
28     }
```

例2：以 `Fun_Declaration` 为例介绍作用域的处理

当碰到函数时，与变量一样，先去符号表中查找函数名称进行验证，如果没有问题，这个时候就会调用 `create()` 函数创建一张新的符号表（这个符号表的作用域只有函数的参数），然后调用 `fs.push()` 方法将当前函数的信息 `Entry` 保存，因为在后续的递归下降验证参数时，需要知道当前处于哪一个函数中。在做好前期准备后紧接着就可以调用其子节点的语义分析驱动函数进行递归下降分析，待子节点全部分析完毕后，调用 `copyParams` 函数将函数的参数（位于当前栈顶的符号表中）弹出，并存储到函数 `Entry` 的 `params` 数组中，然后再将当前符号表从符号表栈中弹出。

需要拷贝函数参数的原因是，函数的参数位于单独的作用域中（这样就可以避免与全局变量冲突），当函数解析完成，弹出作用域符号表时，里面的节点都会被释放，后续如果在处理 `call` 类型的节点验证调用的参数时将会找不到原来的参数列表。

```
1  bool Fun_Declaration(SyntaxTree* root) {
2      bool flag = true;
3      variableType vt = SeUtil::getVariableTypeByValue(root->children[0]-
4      >value);
5      if (vt != VARERROR) {
6          Entry* e = stbl.find(root->children[1]->value);
```

```

6         if (e == NULL) {
7             e = new Entry(Function, vt, root->children[1]->value);
8             stbl.insert(e);
9             stbl.create();
10            fs.push(e); //Subsequent operations are based on the current
function stack
11            flag = parse(root->children[2]) && parse(root->children[3]);
12            fs.pop();
13            e->params.clear();
14            //other wise the table for params will be popped and cannot find
after
15            stbl.copyParams(e);
16            stbl.remove();
17            Entry *ee = stbl.lookup(root->children[1]->value);
18        } else {
19            seUtil::serror(root->children[1],
20                           "function " + root->children[1]->value +
21                           " already declared.");
22            flag = false;
23        }
24    } else {
25        seUtil::serror(root->children[0],
26                       "invalid variable type " + root->children[0]-
>value);
27        flag = false;
28    }
29    root->attr.dataType = VARERROR;
30    return flag;
31 }

```

例3: 以 Relop 为例介绍属性计算的方法

当碰到诸如 ==, > 等关系运算符时, 会先去递归下降地调用子节点的驱动程序, 待子节点完成属性计算后, 检查两个子节点的值类型是否相等, 如果值类型不相等则报错, 并将该节点的值类型设置为 ERROR, 否则将该节点的值类型设置为 Boolean (关系运算的结果是 boolean 类型)

```

1 bool Relop(SyntaxTree* root) {
2     bool flag = parse(root->children[0]) && parse(root->children[1]);
3     if (flag && root->children[0]->attr.dataType != root->children[1]-
>attr.dataType) {
4         seUtil::serror(root,
5                        "Incompatible variable types between operator " +
root->value);
6         flag = false;
7     }
8     if (flag) {
9         root->attr.dataType = Boolean;
10    } else {
11        root->attr.dataType = VARERROR;
12    }
13    return flag;
14 }

```

具体的语义验证规则有:

- 整形不能值与布尔型值相互赋值或者运算
- 仅有布尔型值才能够成为 if 和 while 语句的条件

- 变量使用前必须定义
- 函数调用前必须未定义
- 变量不能重复定义
- 函数不能重复定义
- 赋值号两边类型必须匹配
- 操作数类型必须匹配并且操作数与操作符必须匹配
- `return` 语句返回类型与函数定义的返回类型必须匹配
- 函数调用时实参与形参的数目和类型必须匹配
- 不能对非数组变量使用[...]
- 不能对普通变量使用(...)或()
- 数组访问操作符[...]的下标必须是整型

5. 针对语义分析的测试

在项目的源文件 `/test/semantic/` 目录下包含了语义分析的测试程序，命名为：

- `e[数字编号].txt`：错误例子，如 `e1.txt`
- `c[数字编号].txt`：正确例子，如 `c1.txt`

这里给出部分程序测试的截图：

例1:

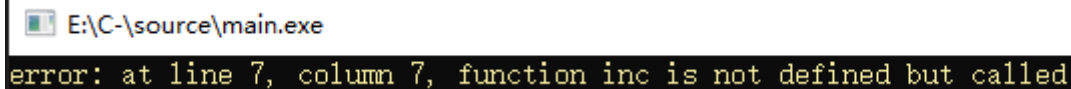
```
1 int main(void) {
2     int x;
3     x = 0;
4     x = i + 1;
5 }
```



The screenshot shows a command prompt window with the file path `E:\C-\source\main.exe`. Below the path, a red error message is displayed: `error: at line 7, column 9, variable i not declared before use`.

例2:

```
1 int main(void) {
2     int i;
3     int j;
4     int i;
5 }
```



The screenshot shows a command prompt window with the file path `E:\C-\source\main.exe`. Below the path, a red error message is displayed: `error: at line 7, column 7, function inc is not defined but called`.

例3:


```

1  int func(int i) {
2      return i;
3  }
4
5  int func(void) {
6      return 0;
7  }
8
9  int main(void) {
10     return 0;
11 }

```

E:\C-\source\main.exe

error: at line 8, column 8, function func already declared.

例4:

```

1  int func(int i) {
2      return i;
3  }
4
5  int main(void) {
6      func(1 == 2);
7  }

```

E:\C-\source\main.exe

error: at line 9, column 8, param 1 requires type Integer but get Boolean

例5:

```

1  int main(void) {
2      int i;
3      i[0];
4  }

```

E:\C-\source\main.exe

error: at line 6, column 5, variable i is not an array

例6:

```

1  int main(void) {
2      int i;
3      i(0);
4  }

```

E:\C-\source\main.exe

error: at line 6, column 5, variable i is not a function

例7:

```
1 int func(int i) {
2     return i;
3 }
4
5 int main(void) {
6     func(1, 2, 3);
7 }
```

E:\C-\source\main.exe

error: at line 10, column 8, function func requires 1 params but get 3

五、中间代码生成和代码优化

中间代码生成

数据结构

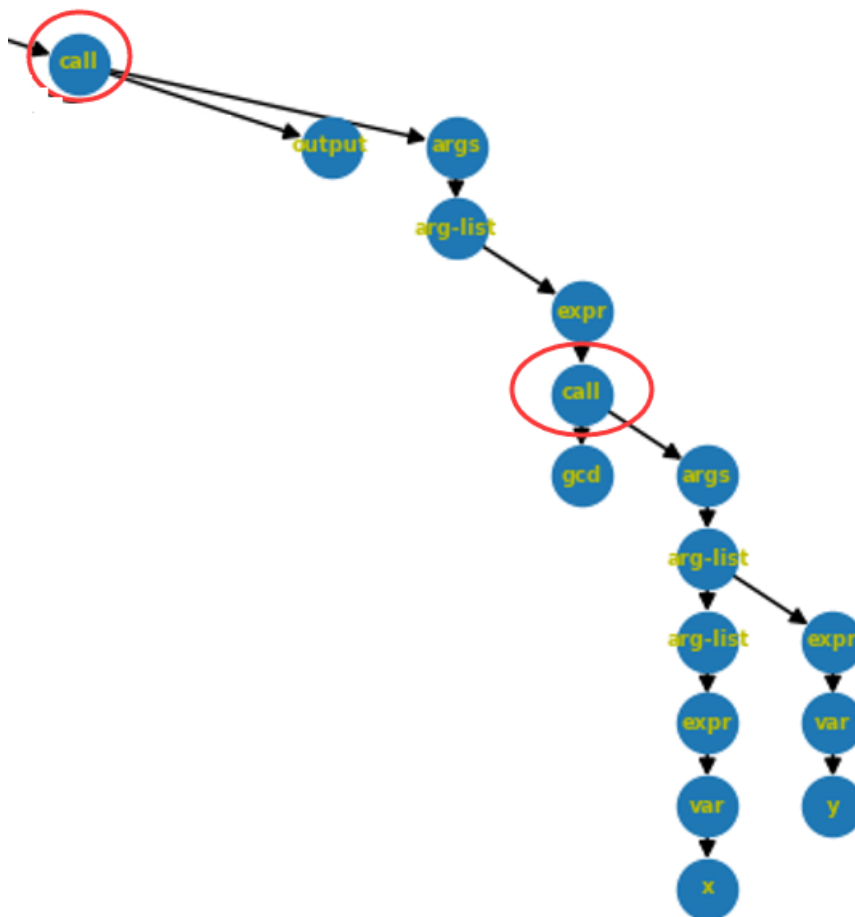
由于我们在生成中间代码之后还需要对代码进行优化，因此在翻译代码时不能直接边翻译边输出，需要用内存将生成的代码记录下来，以便之后的优化。由于c语言是一个函数组成的语言，因此我们将每个函数定义为如下的类。

```
1 class Func{
2 public:
3     string decl;
4     vector<string> params;
5     vector<string> codeblocks;
6     variableType retType;
7     map<string,string> var2reg;
8     Func(){}
9 };
```

- `decl` 记录当前函数的名字
- `params` 记录当前函数需要的参数名
- `codeblocks` 记录该函数的所有语句
- `retType` 标记该函数的返回类型
- `var2reg` 因为源代码和中间码当中对于同一个变量的定义不同，所以需要建立一个对应关系。

函数调用的翻译

函数调用要注意抽象语法树中的Call结点，以如下抽象语法树为例：



call结点下面的第一个儿子是函数名，第二个是args。我们需要获得 ARG x;CALL name 结构的中间代码。因此我们需要分析args。这里就需要递归分析，每次在arg-list的单个孩子或者右孩子里进行递归，最后得到对应的 ARG x。注意在这里由于要进行递归，所以表达式expr的值可能会比参数表达更先被push到codeblock里。因为，本次代码我们使用mips进行目标代码的翻译，mips可以使用寄存器传参，因此此处参数的传递顺序可以和参数申明一样，我们使用vector把当时生成参数先存下来。当然如果要用栈传参数的话，参数声明和传递的顺序应该相反，可以用栈结构储存和提取。

最终生成代码：

```

1  void Arg_List(SyntaxTree* root) {
2      Func* currentFunc;
3      currentFunc = Code[fs.top()->identifier]; //当前栈的Func
4      if (root->children.size() == 1) {
5          string ret = codegen(root->children[0]);
6          //currentFunc->codeblocks.push_back("ARG "+ret);
7          arg_vec.push_back("ARG "+ret);
8      } else if (root->children.size() == 2) {
9          Arg_List(root->children[0]);
10         string ret = codegen(root->children[1]);
11         //currentFunc->codeblocks.push_back("ARG "+ret);
12         arg_vec.push_back("ARG "+ret);
13     }
14 }
15
16 void Args(SyntaxTree* root) {
17     if (root->children.size() == 1) {
18         Arg_List(root->children[0]);
19     }
20     // if Args has no child , do nothing
  
```

```

21     }
22
23     string call(SyntaxTree* root) {
24         Func* currentFunc;
25         currentFunc = Code[fs.top()->identifier]; //当前栈的Func
26         Args(root->children[1]);
27         for(int i=0; i < arg_vec.size(); i++){
28             currentFunc->codeblocks.push_back(arg_vec[i]);
29         }
30         arg_vec.clear(); //清空当前函数参数表
31         if (Code[root->children[0]->value]->retType != Void){
32             string reg = "v"+getNewReg();
33             currentFunc->codeblocks.push_back(reg+" := "+"CALL "+ root->
>children[0]->value);
34             return reg;
35         }else{
36             currentFunc->codeblocks.push_back("CALL "+ root->children[0]-
>value);
37             return "";
38         }
39     }

```

```

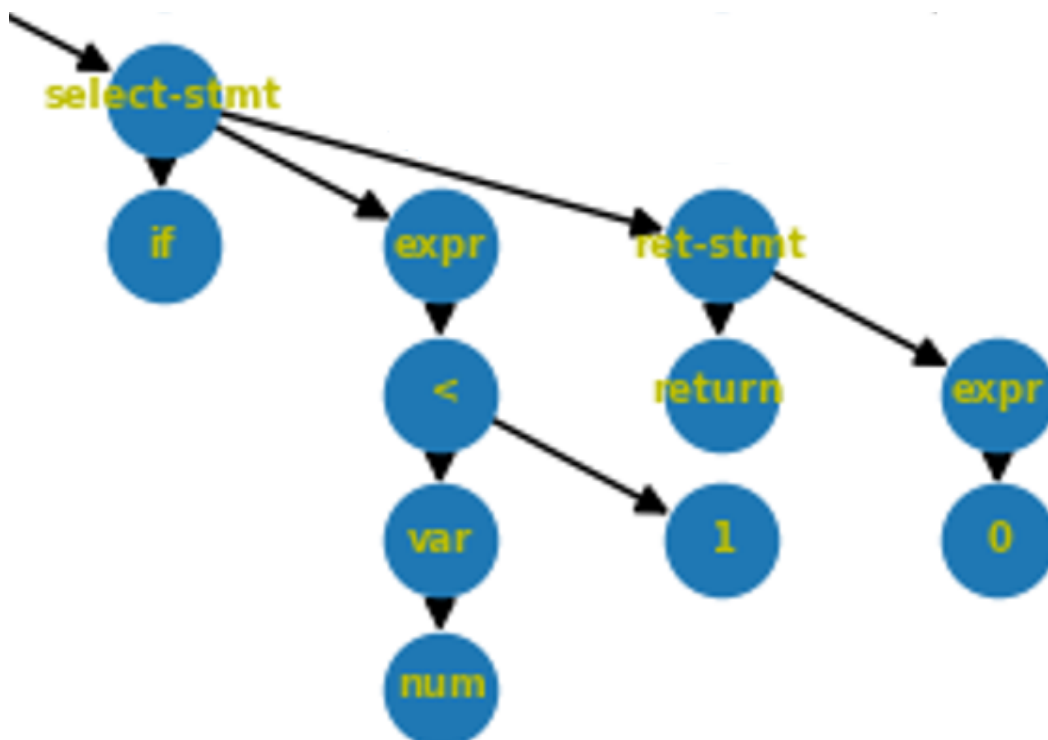
1  ARG v8
2  ARG v9
3  v12 := CALL gcd
4  ARG v12
5  CALL output

```

if else 的翻译

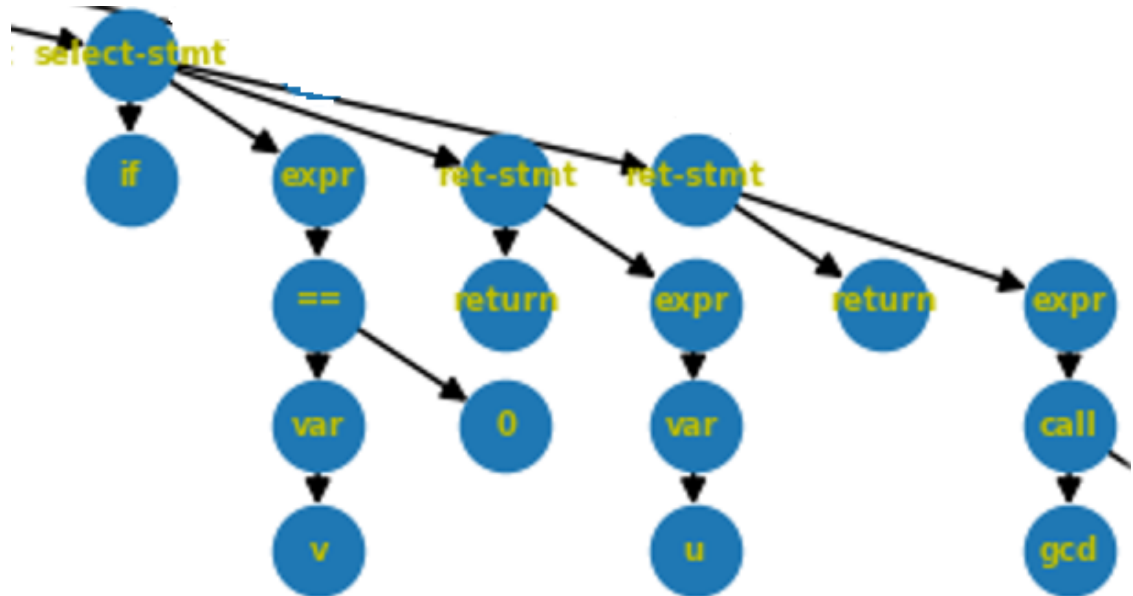
if else 有两种情况，分别对应两种树的结构图

- 没有else



没有else的情况我们可以把代码分成两个代码块，如果满足条件就顺序执行，否则就跳转执行接下来的代码，因此只需要两个label就可以划分。

- 有else



```

1 IF condition GOTO label1
2 GOTO label2
3 LABEL label1:
4 /* if statements */
5 GOTO label3
6 LABEL label2:
7 /* else statements */
8 LABEL label3:
9 /* other code */
  
```

这里其实还要考虑到/* other code*/ 可能为空，这时候的label3又会变的不必要。这一步会在代码优化部分进行，因为进行中间代码分析时需要有比较universal的方法去分析。

```

1 void Selection Stmt(SyntaxTree* root) {
2     Func* currentFunc;
3     currentFunc = Code[fs.top()->identifier]; //当前栈的Func
4     string label1 = getLabelName();
5     string label2 = getLabelName();
6     string label3 = getLabelName();
7     string condition = codegen(root->children[1]);
8     currentFunc->codeblocks.push_back("IF "+condition+" GOTO "+label1);
9     currentFunc->codeblocks.push_back("GOTO "+label2); //GOTO label2
10    currentFunc->codeblocks.push_back("LABEL "+label1+ " :");
11    string stmt1 = codegen(root->children[2]);
12    if (root->children.size()==4){
13        currentFunc->codeblocks.push_back("GOTO "+label3);
14    }
15    currentFunc->codeblocks.push_back("LABEL "+label2+ " :");
16    if (root->children.size()==4){
17        string stmt2 = codegen(root->children[3]);
18        currentFunc->codeblocks.push_back("LABEL "+label3+ " :");
19    }
20 }
  
```

表达式的翻译

表达式是中间代码中最常见的部分，但也算是最简单分析。只需将左右子节点递归生成，最后用中间的结点符号进行计算。因为子节点的类型可能也是expression，所以他们需要返回一个string类型的字符串表示当前存储该节点的值。

以relation-operator 为例：

```
1      string Relop(SyntaxTree* root) { //relation-operator
2          Func* currentFunc;
3          currentFunc = Code[fs.top()->identifier]; //当前栈的Func
4          string a = codegen(root->children[0]);
5          string b = codegen(root->children[1]);
6          string retreg = "v" + getNewReg();
7          currentFunc->codeblocks.push_back(retreg + " := " + a + " " + root-
>value+ " " + b);
8          cout << retreg + " := " + a + root->value + b << endl;
9          return retreg;
10     }
```

中间代码生成的优化

由于生成的中间码用的方法会universal，显得整个中间码很累赘甚至可能会产生小错误。因此，我们需要对代码进行优化。

我们的优化主要包括以下四个方面：

- 将 `v8 := CALL input; v6 := v8` 类型的语句合成 `v6 := CALL input`。
- 如果表达式的左右结点都是立即数，则在编译同时直接算出结果
- 将 `v1 := v0 == #1 ; IF v1 GOTO label0` 类型的语句合成 `IF v0 == #1 GOTO label0`。
- 修复了当if return-stmt else return-stmt 结构的函数时会出现label3跑到函数之外的bug

附上我们的代码优化函数的源码：

```
1      void CodeOptimizer(){
2          for(int i=0; i < FuncDeclOrder.size(); i++)
3          {
4              vector<int> delindex;
5              Func* t = Code[FuncDeclOrder[i]];
6              cout << "Debug: Current Func : " << FuncDeclOrder[i] << endl;
7              cout << "Debug: number of instructions : " << t-
>codeblocks.size() << endl;
8              if( t->codeblocks.size() < 2) continue;
9              for(int j=0; j < t->codeblocks.size()-1; j++)
10             {
11                 vector<string> current = split(t->codeblocks[j], " ");
12                 vector<string> next = split(t->codeblocks[j+1], " ");
13                 /*
14                     v8 := CALL input \
15                     v6 := v8          /   v6 := CALL input
16                 */
17                 if(next.size()==3 && next[2]==current[0]){
18                     t->codeblocks[j] = next[0];
19                     for(int k=1; k < current.size(); k++)
20                         t->codeblocks[j] += " " + current[k];
21                     delindex.push_back(j+1);
22                     cout<<"NEED to DEL :" << t->codeblocks[j+1] <<endl;
```

```

23         //t->codeblocks.erase(t->codeblocks.begin()+j+1);
24     }
25
26     //如果有俩立即数 直接算出来
27     if(current.size() == 5 && current[2].at(0) == '#' &&
current[4].at(0) == '#')
28     {
29         int a =
atoi(current[2].substr(1,current[2].size()).c_str());
30         int b =
atoi(current[4].substr(1,current[2].size()).c_str());
31         int rs;
32         if (current[3] == "+")
33             rs = a+b;
34         else if (current[3] == "-")
35             rs = a-b;
36         else if (current[3] == "*")
37             rs = a*b;
38         else if (current[3] == "/")
39             rs = a/b;
40         t->codeblocks[j] = current[0]+" " + current[1] + " #" +
to_string(rs);
41     }
42
43     /*
44         v1 := v0 == #1      \
45         IF v1 GOTO label0 /   IF v0 == #1 GOTO label0
46     */
47     if(next[0]=="IF" && next[1]==current[0]){
48         t->codeblocks[j] = "IF";
49         for(int k=2; k < current.size(); k++)
50             t->codeblocks[j] += " " + current[k];
51         t->codeblocks[j] += " GOTO "+next[3];
52         delindex.push_back(j+1);
53         cout<<"NEED to DEL :" << t->codeblocks[j+1] <<endl;
54         //t->codeblocks.erase(t->codeblocks.begin()+j+1);
55     }
56
57     // if else 时如果label出现func最后 则删除该label
58     if (j+1 == t->codeblocks.size()-1 && next[0]=="LABEL"){
59         //cout<<"Debug: " << t->codeblocks[j+1] <<endl;
60         for(int m=0;m<t->codeblocks.size();m++){
61             if(t->codeblocks[m].find(next[1]) != string::npos){
62                 cout<<"NEED to DEL :" << t->codeblocks[m]
<<endl;
63                 delindex.push_back(m);
64             }
65         }
66     }
67 }
68 sort(delindex.begin(),delindex.end());
69 // 删除指定下标的元素
70 int count = 0;
71 for ( vector<string>::iterator it = t->codeblocks.begin(); it
!= t->codeblocks.end(); )
72 {
73     int m=0;
74     for(m=0; m < delindex.size();m++){

```

```

75         //cout<<"Debug: delindex[m]:" << delindex[m] <<
"count:" << count <<endl;
76         if ( count == delindex[m] ){
77             cout << "DEL: " << (*it) << endl;
78             it = t->codeblocks.erase( it );
79
80             break;
81         }
82     }
83     if(m != delindex.size()){
84         count ++;
85     }else{
86         ++it;
87         count++;
88     }
89 }
90
91 }
92 }

```

以该代码为例：

```

1  int fact(int n)
2  {
3      if (n == 1 ){
4          return n;
5      }else{
6          return (n * fact(n-1));
7      }
8  }
9
10 int input(void) { return 2; }
11 void output(int x) {}
12
13 int main(void)
14 {
15     int m;
16     int result;
17     m = input();
18     if(m>1)
19         result = fact(m);
20     else
21         result = 1;
22     output(result);
23     return 0;
24 }

```


out > ≡ InterCode_out.txt	out_test > ≡ c4_IR.txt
1 FUNCTION fact :	FUNCTION fact :
2 PARAM v0	PARAM v0
3 v1 := v0 == #1	IF v0 == #1 GOTO label0
4 IF v1 GOTO label0	GOTO label1
5 GOTO label1	LABEL label0 :
6 LABEL label0 :	RETURN v0
7 RETURN v0	LABEL label1 :
8 GOTO label2	v2 := v0 - #1
9 LABEL label1 :	ARG v2
10 v2 := v0 - #1	v3 := CALL fact
11 ARG v2	v4 := v0 * v3
12 v3 := CALL fact	RETURN v4
13 v4 := v0 * v3	
14 RETURN v4	
15 LABEL label2 :	
16	FUNCTION input :
17 FUNCTION input :	RETURN #2
18 RETURN #2	
19	
20 FUNCTION output :	FUNCTION output :
21 PARAM v5	PARAM v5
22	
23 FUNCTION main :	FUNCTION main :
24 v8 := CALL input	v6 := CALL input
25 v6 := v8	IF v6 > #1 GOTO label3
26 v9 := v6 > #1	GOTO label4
27 IF v9 GOTO label3	LABEL label3 :
28 GOTO label4	ARG v6
29 LABEL label3 :	v7 := CALL fact
30 ARG v6	GOTO label5
31 v10 := CALL fact	LABEL label4 :
32 v7 := v10	v7 := #1
33 GOTO label5	LABEL label5 :
34 LABEL label4 :	ARG v7
35 v7 := #1	CALL output
36 LABEL label5 :	RETURN #0
37 ARG v7	
38 CALL output	
39 RETURN #0	

左边是没加代码优化生成的IR，可以明显看出左边代码会显得更累赘，代码的优化提升了运行的效率。同时可以关注到左边代码的第15行，由于if else结构中缺失了前面提到的other code。因此，出现了程序上的bug。而代码优化端就没有这个问题。

六、目标代码生成和运行时环境

该部分采用Python来进行翻译，并在mips虚拟机下成功运行。

input 和 output的写入

考虑到测试编译器的时候要大量使用输入输出，我们将这两个函数直接以mips指令的方式写好载入汇编文件之中。

```

1 .data
2 _prompt: .asciiz "Enter an integer:"
3 _ret: .asciiz "\\n"
4 .globl main
5 .text

```

```

6  input:
7      li $v0,4
8      la $a0,_prompt
9      syscall
10     li $v0,5
11     syscall
12     jr $ra
13 output:
14     li $v0,1
15     syscall
16     li $v0,4
17     la $a0,_ret
18     syscall
19     move $v0,$0
20     jr $ra

```

寄存器的分配

因为汇编的寄存器有限，因此我们在翻译中间码前，先把中间码中所有的变量过一遍，储存他们的名字（同一个变量出现几次存几次）。当运行到该变量时就添加一个和寄存器的映射。当某个变量不在出现，则它对应的寄存器也释放了。

相关代码如下：

```

1  regs=
   ['t1','t2','t3','t4','t5','t6','t7','t8','t9','s0','s1','s2','s3','s4','s5',
   's6','s7']
2  table={}          #存映射关系
3  reg_ok={}        #存寄存器是否可用
4  variables=[]     #同一个变量多次存入
5
6  def Load_Var(Inter):
7      global variables
8      temp_re='(v\d+)'
9      for line in Inter:
10         temps=re.findall(temp_re,' '.join(line))
11         variables+=temps
12  def Get_R(string):
13      try:
14         variables.remove(string) # 每用到一次就删一次 一旦删完表明这个变量绑定的寄存
器可以释放
15     except:
16         pass
17     if string in table:
18         return '$'+table[string] #如果已经存在寄存器分配，那么直接返回寄存器
19     else:
20         for key in table.keys():          #当遇到未分配寄存器的变量时，清空之前所
有分配的临时变量的映射关系!!!
21             if 'v' in key and key not in variables:
22                 reg_ok[table[key]]=1
23                 del table[key]
24         for reg in regs:                  #对于所有寄存器
25             if reg_ok[reg]==1:            #如果寄存器可用
26                 table[string]=reg        #将可用寄存器分配给该变量，映射关系存到table中
27                 reg_ok[reg]=0            #寄存器reg设置为已用
28         return '$'+reg

```

代码翻译

将中间码——对应翻译为mips汇编。注意函数调用时的规则。详细代码可以看源文件。

七、整体代码运行效果

以 `test/semantic /c1.txt` 求两数的最大公约数为例:

生成的中间码:

```
1  FUNCTION input :
2  RETURN #1
3
4  FUNCTION output :
5  PARAM v0
6
7  FUNCTION gcd :
8  PARAM v1
9  PARAM v2
10 IF v2 == #0 GOTO label0
11 GOTO label1
12 LABEL label0 :
13 RETURN v1
14 LABEL label1 :
15 v4 := v1 / v2
16 v5 := v4 * v2
17 v6 := v1 - v5
18 ARG v2
19 ARG v6
20 v7 := CALL gcd
21 RETURN v7
22
23 FUNCTION main :
24 v8 := CALL input
25 v9 := CALL input
26 ARG v8
27 ARG v9
28 v12 := CALL gcd
29 ARG v12
30 CALL output
31
```

生成的汇编码:

```
1
2 .data
3 _prompt: .asciiz "Enter an integer:"
4 _ret: .asciiz "\n"
5 .globl main
6 .text
7 input:
8     li $v0,4
9     la $a0,_prompt
10    syscall
11    li $v0,5
12    syscall
13    jr $ra
```

```

14  output:
15      li $v0,1
16      syscall
17      li $v0,4
18      la $a0,_ret
19      syscall
20      move $v0,$0
21      jr $ra
22  gcd:
23      li $t1,0
24      beq $a1,$t1,label0
25      j label1
26  label0:
27      move $v0,$a0
28      jr $ra
29  label1:
30      div $a0,$a1
31      mflo $t2
32      mul $t3,$t2,$a1
33      sub $t2,$a0,$t3
34      move $t0,$a0
35      move $a0,$a1
36      move $t1,$a1
37      move $a1,$t2
38      addi $sp,$sp,-24
39      sw $t0,0($sp)
40      sw $ra,4($sp)
41      sw $t1,8($sp)
42      sw $t2,12($sp)
43      sw $t3,16($sp)
44      sw $t4,20($sp)
45      jal gcd
46      lw $a0,0($sp)
47      lw $ra,4($sp)
48      lw $a1,8($sp)
49      lw $t2,12($sp)
50      lw $t3,16($sp)
51      lw $t4,20($sp)
52      addi $sp,$sp,24
53      move $t2 $v0
54      move $v0,$t2
55      jr $ra
56  main:
57      addi $sp,$sp,-4
58      sw $ra,0($sp)
59      jal input
60      lw $ra,0($sp)
61      move $t2,$v0
62      addi $sp,$sp,4
63      addi $sp,$sp,-4
64      sw $ra,0($sp)
65      jal input
66      lw $ra,0($sp)
67      move $t3,$v0
68      addi $sp,$sp,4
69      move $t0,$a0
70      move $a0,$t2
71      move $t1,$a1

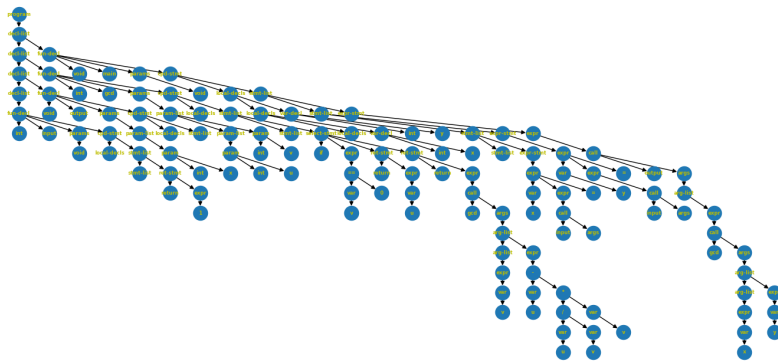
```

```

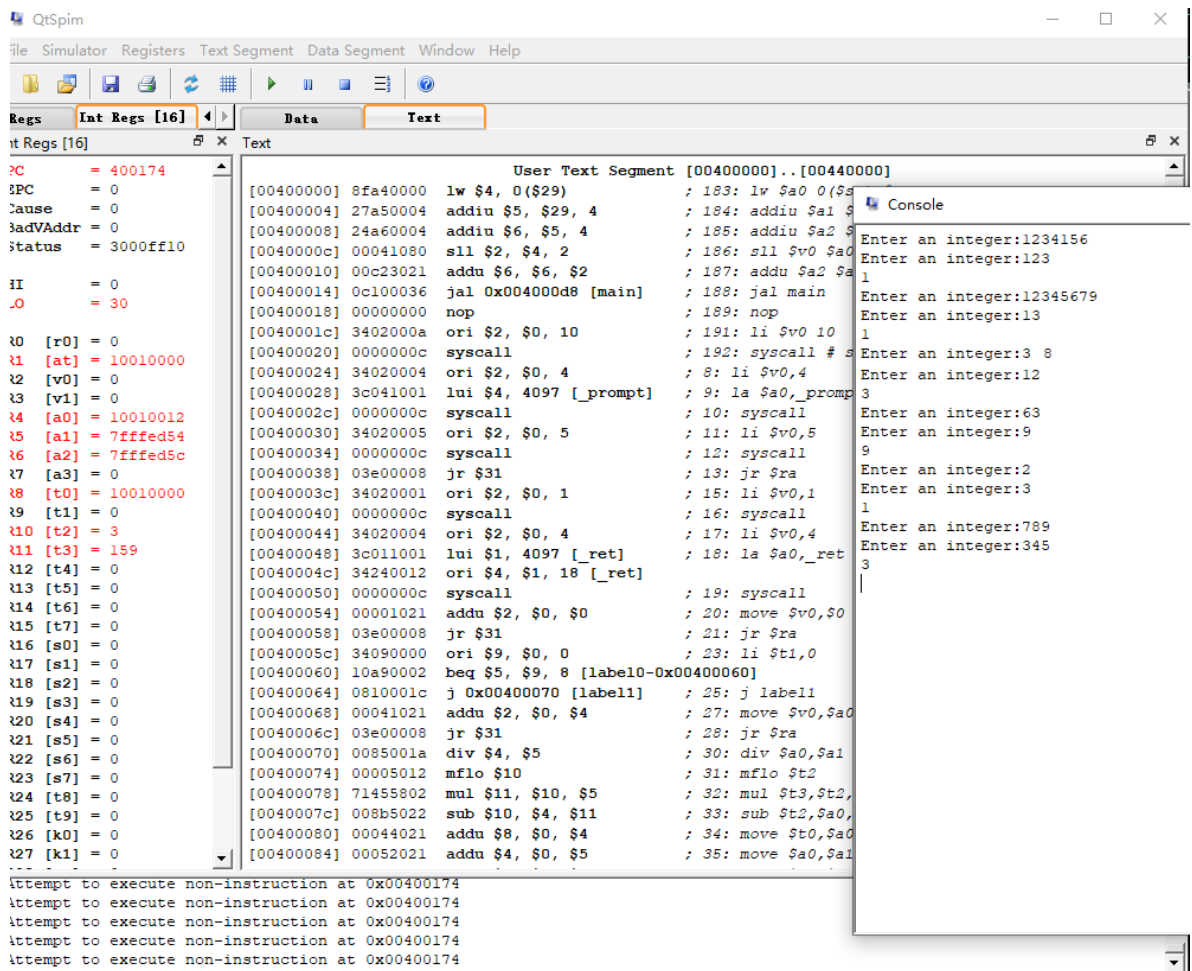
72      move $a1,$t3
73      addi $sp,$sp,-24
74      sw $t0,0($sp)
75      sw $ra,4($sp)
76      sw $t1,8($sp)
77      sw $t2,12($sp)
78      sw $t3,16($sp)
79      sw $t4,20($sp)
80      jal gcd
81      lw $a0,0($sp)
82      lw $ra,4($sp)
83      lw $a1,8($sp)
84      lw $t2,12($sp)
85      lw $t3,16($sp)
86      lw $t4,20($sp)
87      addi $sp,$sp,24
88      move $t2 $v0
89      move $t0,$a0
90      move $a0,$t2
91      addi $sp,$sp,-4
92      sw $ra,0($sp)
93      jal output
94      lw $ra,0($sp)
95      addi $sp,$sp,4
96

```

生成的抽象语法树图：



在mips虚拟机下的运行结果：



八、附录

1. 代码目录结构

- `header`：头文件
- `out`：语法树输出
- `parser`：Lex和Yacc源文件
- `source`：源代码目录
- `test`：测试用例
- `main.exe` 主目录下已经编译完成的可执行文件

2. 代码的运行方式

所有程序在Windows系统下利用vscode编写，使用g++，bison，lex等进行编译，其中抽象语法树可视化和目标代码生成部分使用了Python。

在主目录下运行 `mk.sh`，进行lex和bison分析

运行 `g++ source/main.cpp -o main.exe` 编译出可执行文件

运行 `./main.exe FILENAME` (FILENAME为合法的c-源文件)，生成 `out/InterCode_out.txt` 和 `out/SyntaxTree_out.txt`

`python source/DrawSyntaxTree.py` 可以画出当前 `out/SyntaxTree_out.txt` 所表示的语法树

`python source/ObjCodeGen.py` 可以生成当前中间代码 `out/InterCode_out.txt` 所表示的mips 汇编码