1 支持语法

1.1 语法

本编译器实现的是一种类C的语言，在C语言的语法上进行删减，支持的语法包括（类型以int示例，float、char同理）：

语法	描述
int a;	变量定义
int a=1；	含初始值的变量定义
int a[1]；	数组定义
int a[1]=1;	含初始化的数组定义
int f(){}	函数定义
int f(int a){}	带参数的函数定义
int f(int a=1){}	带参数和缺省值的函数定义
a = 1；	赋值语句
a = 1+2；	右侧表达式的赋值语句
(1+2)	括号规定优先级
(int)1.2	强制类型转换
if(condition){stmt1} else{stmt2}	if语句（含else）
if(condition){stmt}	if语句（不含else）
while(condition){stmt}	while语句
do{stmt}while(condition);	do-while语句
for(exp;condition;exp){stmt3}	for语句
a = f(1);	函数调用
return a；	返回值
return ；	void型返回

1.2 运算符

运算符的优先级与C语言规定的运算符优先级一致。

支持的单目运算符包括：

```
++，--，!，~，+，-，&，*
```

支持的双目运算符包括：

```
<<=，>>=，<<，>>，==，!=，<=，>=，->，+=，-=，*=，/=，%=，&=，|=，^=，
&&，||，|，&，<，>，=，*，/，%，^
```

支持的三目运算符包括：

```
? :
```

1.3 特殊说明

1.3.1 函数声明

函数声明对于汇编代码的生成没有影响，所以并没有实现，因为在语义分析部分允许函数调用后声明的函数，所以实际上并没有对C语言的语法产生影响。

1.3.2 "{ }"的管理

大括号在yacc分析时作为分割代码块使用，但仅仅在函数定义、if、else、while、do、for后有意义，且在if、while、do、for后可以省略，否则大括号没有意义。

1.3.3 循环语句

if、while、for的条件语句都可以为空，for的三条语句都可以为空。不与if配套的else语句不能编译。

1.3.4 数组

出于简便实现，同时考虑到多维数组可以通过固定的方式转化为多个二维数组，故本编译器仅实现到二维数组。

1.3.5 指针

考虑到我们已经实现了取地址运算（&）以及按地址取值运算（*），同时为了减少对内存的随意访问，故没有允许指针变量。

2 词法分析

2.1 正则表达式

2.1.1 目标

词法分析阶段将输入的文件中的字符串进行切割，转化为约定的token，最后将形成的token流传递给语法分析器。
本阶段我们组借助lex进行词法分析。

1.1.2 正则表达式定义

定义了本编译器接受的正则表达式，在接收到对应的字符串时通过定义的正则表达式生成对应的token。

```
IDENTIFIER ([a-zA-Z]|'_')([a-zA-Z]|'_'|[0-9])*
FLOATNUM ([0-9]+)\.([0-9]+)
INTNUM [0-9]+
CHARACTERS \'[\x00-\x7F]\'


LEFT_ASSIGN "<<="
RIGHT_ASSIGN ">>="
```

```
EQUAL "=="
NOTEQUAL "!="
LESSEQUAL "<="
GREATEQUAL ">="

increment "++"
decrement "--"
arrow "->"

ADD_ASSIGN "+="
SUB_ASSIGN "-="
MUL_ASSIGN "*="
DIV_ASSIGN "/="
MOD_ASSIGN "%="
AND_ASSIGN "&="
OR_ASSIGN "|="
XOR_ASSIGN "^="

LEFT_SHIFT "<<"
RIGHT_SHIFT ">>"

AND "&&"
OR "||"

ORBIT "|"
ANDBIT "&"

LESSTHAN "<"
GREATTHAN ">"
PURE_ASSIGN "="


COMMENT "/*"([^*]|\*+[^*/])*\*+"/"
```

2.2 数据结构

词法分析将token转化为约定的语法树节点，其数据结构为

```
`struct TreeNode{`
    `int row;                    //行数`
    `string type;                //类型`
    `string value;               //字符值`
    `string scope;               //生命周期`
    `struct Attr attr;           //属性`
    `vector<TreeNode *> child; //子节点
    TreeNode* prt;`
`}`
```

其中，属性的数据结构为

```
struct Attr {
    Type type;  //类型，只有函数和变量有
    int numval; //数值
    char cnumval;
    float fnumval;
    int isdone;
};
```

2.3 原理及实现

2.3.1 符号处理

在接收到符号时，不进行处理，直接向后传递到语法分析，在语法分析阶段直接舍弃，不进入语法树节点。

```
";"     {
    yylval = nullptr;
    return *yytext;
}

","     {
    yylval = nullptr;
    return *yytext;
}

"("     {
    yylval = nullptr;
    return *yytext;
}

")"     {
    yylval = nullptr;
    return *yytext;
}

"["     {
    yylval = nullptr;
    return *yytext;
}

"]"     {
    yylval = nullptr;
    return *yytext;
}

"{"     {
    yylval = nullptr;
    return *yytext;
}

"}"     {
    yylval = nullptr;
    return *yytext;
}

"&"     {
    yylval = nullptr;
    return *yytext;
}

"+"     {
    yylval = nullptr;
    return *yytext;
}
```

```
"-"     {
    yylval = nullptr;
    return *yytext;
}

"*"     {
    yylval = nullptr;
    return *yytext;
}

"/"     {
    yylval = nullptr;
    return *yytext;
}

"%"     {
    yylval = nullptr;
    return *yytext;
}


"~"     {
    yylval = nullptr;
    return *yytext;
}

"!"     {
    yylval = nullptr;
    return *yytext;
}

"?"     {
    yylval = nullptr;
    return *yytext;
}

":"     {
    yylval = nullptr;
    return *yytext;
}
```

2.3.2 关键字

在接收到C语言关键字时，需要新建语法树节点并插入语法树，记录该关键字的代码行数、类型和值，最后返回规定的token。

```
"if"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "IF";
  return IF;
}

"else"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "ELSE";
  return ELSE;
}

"while"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "WHILE";
  return WHILE;
}

"for" {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "FOR";
  return FOR;
}

"do" {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "DO";
  return DO;
}

"int"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "INT";
  return INT;
}

"char"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "CHAR";
  return CHAR;
}

"float"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "FLOAT";
  return FLOAT;
}



"void"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "VOID";
  return VOID;
}

"return"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "RETURN";
  return RETURN;
}
```

```
"const"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "CONST";
  return CONST;
}

"volatile"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "VOLATILE";
  return VOLATILE;
}

"break"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "BREAK";
  return BREAK;
}

"continue"   {
  yylval = new TreeNode;
  yylval->row = cur_row;
  yylval->type = "KEYWORD";
  yylval->value = "CONTINUE";
  return CONTINUE;
}
```

2.3.3 C语言标识符、数字、字符处理

在接收到C语言的标识符、数字或字符时，新建语法树节点并插入语法树，记录该标识符或数字或字符的代码行数、类型、值，最后返回规定的token。其中，值（标识符的唯一标识、数字的值、字符的值）统一以字符串的形式保存。

```
{IDENTIFIER}    {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "IDENTIFIER";
    yylval->value = yytext;
    return IDENTIFIER;
}

{INTNUM}    {
    yylval = new TreeNode;
    yylval->row = cur_row;
    //yylval->col = cur_column;
    yylval->type = "INTNUM";
    yylval->value = yytext;
    yylval->attr.type= Type::Int;
    yylval->attr.numval = atoi(yytext);
    return INTNUM;
}

{FLOATNUM}    {
    yylval = new TreeNode;
    yylval->row = cur_row;
    //yylval->col = cur_column;
    yylval->type = "FLOATNUM";
    yylval->value = yytext;
    yylval->attr.type= Type::Float;
    yylval->attr.fnumval = atoi(yytext);
    return FLOATNUM;
}

{CHARACTERS} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    //yylval->col = cur_column;
    yylval->type = "CHARACTERS";
    yylval->value = yytext;
    yylval->attr.type= Type::Char;
    yylval->attr.cnumval = atoi(yytext);
    return CHARACTERS;
}
```

2.3.4 C语言运算符处理

在接收到C语言运算符之后，新建语法树节点并插入语法树，记录该运算符的代码行数、类型、值，最后返回规定的token。

```
{LEFT_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "LEFT_ASSIGN";
    yylval->value = yytext;
    return LEFT_ASSIGN;
}

{RIGHT_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "RIGHT_ASSIGN";
    yylval->value = yytext;
    return RIGHT_ASSIGN;
}

{EQUAL} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "EQUAL";
    yylval->value = yytext;
    return EQUAL;
}

{NOTEQUAL} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "NOTEQUAL";
    yylval->value = yytext;
    return NOTEQUAL;
}

{LESSEQUAL} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "LESSEQUAL";
    yylval->value = yytext;
    return LESSEQUAL;
```

```
}

{GREATEQUAL} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "GREATEQUAL";
    yylval->value = yytext;
    return GREATEQUAL;
}

{increment} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "increment";
    yylval->value = yytext;
    return increment;
}

{decrement} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "decrement";
    yylval->value = yytext;
    return decrement;
}

{arrow} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "arrow";
    yylval->value = yytext;
    return arrow;
}

{ADD_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "ADD_ASSIGN";
    yylval->value = yytext;
    return ADD_ASSIGN;
}

{SUB_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "SUB_ASSIGN";
    yylval->value = yytext;
    return SUB_ASSIGN;
}
{MUL_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "MUL_ASSIGN";
    yylval->value = yytext;
    return MUL_ASSIGN;
}
{DIV_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "DIV_ASSIGN";
    yylval->value = yytext;
    return DIV_ASSIGN;
}
{MOD_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "MOD_ASSIGN";
    yylval->value = yytext;
    return MOD_ASSIGN;
}
{AND_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "AND_ASSIGN";
    yylval->value = yytext;
    return AND_ASSIGN;
}

{OR_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "OR_ASSIGN";
    yylval->value = yytext;
    return OR_ASSIGN;
}
```

```
{XOR_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "XOR_ASSIGN";
    yylval->value = yytext;
    return XOR_ASSIGN;
}

{LEFT_SHIFT} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "LEFT_SHIFT";
    yylval->value = yytext;
    return LEFT_SHIFT;
}

{RIGHT_SHIFT} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "RIGHT_SHIFT";
    yylval->value = yytext;
    return RIGHT_SHIFT;
}

{AND} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "AND";
    yylval->value = yytext;
    return AND;
}

{OR} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "OR";
    yylval->value = yytext;
    return OR;
}

{ANDBIT} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "ANDBIT";
    yylval->value = yytext;
    return ANDBIT;
}

{ORBIT} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "ORBIT";
    yylval->value = yytext;
    return ORBIT;
}

{LESSTHAN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "LESSTHAN";
    yylval->value = yytext;
    return LESSTHAN;
}

{GREATTHAN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "GREATTHAN";
    yylval->value = yytext;
    return GREATTHAN;
}

{PURE_ASSIGN} {
    yylval = new TreeNode;
    yylval->row = cur_row;
    yylval->type = "PURE_ASSIGN";
    yylval->value = yytext;
    return PURE_ASSIGN;
}
```

2.3.5 行数计算

定义全局变量cur_row，当接收到换行符'\n'时，自动递增；对于每一个合法的token，将当前的行数储存在语法树节点中；对于语法报错，报出错误所在的具体行数。

```
\n          { cur_row++; }
```

2.3.6 非法字符处理

当接收到非法字符，打印出对应的行数和字符。

```
[^ \n\t]     { printf("Error at line %d: Mysterious characters
\'%s\'\n",cur_row,yytext); }
```

3 语法分析

3.1 语法树结构

语法树的每个节点需要储存节点的代码函数、节点类型、节点字符值、字符属性、该节点的子节点。

```
class TreeNode {
  public:
    TreeNode() {}
    ~TreeNode() {}
    TreeNode(string t_value) {
        type = "GENERATOR";
        value = t_value;
        attr.type = Void;
        attr.isdone = 0;
        attr.numval = 0;
        attr.fnumval = 0;
        attr.cnumval = 0;
    }
    void setNode(int r, string v, enum Type ty, string nv){
        row = r;
        value = v;
        attr.type = ty;
        if(ty == Char){
            type = "CHARACTER";
            attr.cnumval = nv.at(0);
        }
    }
    TreeNode *AddNode(TreeNode *node) {
        child.push_back(node);
        return this;
    }

    int row;
    string type;              //类型
    string value;             //字符值
    string scope;             //有效范围（所属函数名，全局变量则是“$”）
    struct Attr attr;         //属性
    vector<TreeNode *> child; //子节点
};
```

3.2 文法描述

本编译器接收的所有token包括

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%token increment decrement arrow
%token IDENTIFIER INTNUM FLOATNUM CHARACTERS
%token INT FLOAT CHAR VOID
%token CONST VOLATILE
%token PURE_ASSIGN ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
MOD_ASSIGN
%token LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
%token EQUAL NOTEQUAL LESSEQUAL GREATEQUAL LESSTHAN GREATTHAN AND
OR LEFT_SHIFT RIGHT_SHIFT ANDBIT ORBIT
%token IF DO WHILE FOR SWITCH BREAK CONTINUE RETURN
```

```
%right PURE_ASSIGN ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
MOD_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
%left '+' '-'
%left '*' '/'
```

本编译器的文法从program开始，通过上下文无关文法的规则来实现不同符号优先级，通过递归的方式实现对程序的逐层分析。

特殊地，出于简化语法树的目的，如果CFG的某条规则推导只产生了单个子节点，那么直接合并该节点和子节点（$$=$1），即该规则不会影响所在分支的深度，从而尽量减少在遍历语法树的过程中对无意义节点的访问。

对于一般的规则，需要根据推导出的终结符或非终结符生成新的语法树节点并将其压入本节点的子节点序列中。

```
program : declaration_list {
    // printf("WTF");
    root=new TreeNode("program");
    $$=root;
    $$->AddNode($1);
    $$->row = $1->row;
}
;

declaration_list : external_declaration {
            $$=new TreeNode("declaration_list");
            $$->AddNode($1);
            $$->row = $1->row;

        }
        | declaration_list external_declaration {
            $$=new TreeNode("declaration_list");
            $$->AddNode($1)->AddNode($2);
            $$->row = $1->row;
        }
        ;

external_declaration : declaration {
                        $$ = new
TreeNode("external_declaration_var");
                        $$->AddNode($1);
                        $$->row = $1->row;
                    }
                     | function_definition {
                        $$ = new
TreeNode("external_declaration_fun");
                        $$->AddNode($1);
                        $$->row = $1->row;
                     }
                     ;

declaration : type_specifier init_declarator_list ';' {
                $$ = new TreeNode("declaration");
                $$->AddNode($1)->AddNode($2);
                $$->row = $1->row;
            }
            ;

init_declarator_list : init_declarator {
                        $$ = new TreeNode("init_declarator_list");
                        $$->AddNode($1);
                        $$->row = $1->row;
                    }
                     | init_declarator_list ',' init_declarator {
                        $$ = new TreeNode("init_declarator_list");
                        $$->AddNode($1)->AddNode($3);
                        $$->row = $1->row;
                     }
                     ;

init_declarator : declarator {
                    $$ = new TreeNode("init_declarator");
                    $$->AddNode($1);
                    $$->row = $1->row;
                }
                | declarator PURE_ASSIGN initializer {
                    $$ = new TreeNode("init_declarator");
                    $$->AddNode($1)->AddNode($2)->AddNode($3);
                    $$->row = $1->row;
                }
                ;

type_specifier : INT {
                    $$ = new TreeNode("type_specifier_int");
```

```
                        $$->AddNode($1);
                        $$->row = $1->row;
                    }
                  | FLOAT {
                        $$ = new TreeNode("type_specifier_float");
                        $$->AddNode($1);
                        $$->row = $1->row;
                  }
                  | CHAR {
                        $$ = new TreeNode("type_specifier_char");
                        $$->AddNode($1);
                        $$->row = $1->row;
                  }
                  | VOID {
                        $$ = new TreeNode("type_specifier_void");
                        $$->AddNode($1);
                        $$->row = $1->row;
                  }
                  ;

declarator : direct_declarator {
                $$ = new TreeNode("declarator");
                $$->AddNode($1);
                $$->row = $1->row;
            }
           | pointer direct_declarator {
                $$ = new TreeNode("declarator");
                $$->AddNode($1)->AddNode($2);
                $$->row = $1->row;
           }
           ;


direct_declarator : IDENTIFIER {
                        $$ = new TreeNode("direct_declarator");
                        $$->AddNode($1);
                        $$->row = $1->row;
                    }
                  | direct_declarator '[' INTNUM ']' {
                        $$ = new TreeNode("direct_declarator");
                        $$->AddNode($1)->AddNode($3);
                        $$->row = $1->row;
                  }
```

```
                    ;


pointer : '*' type_qualifier_list {
            $$ = new TreeNode("pointer");
            $$->AddNode($2);
            $$->row = $2->row;
        }
        ;

type_qualifier_list : type_qualifier {
                        $$ = new TreeNode("type_qualifier_list");
                        $$->AddNode($1);
                        $$->row = $1->row;
                    }
                    | type_qualifier_list type_qualifier {
                        $$ = new TreeNode("type_qualifier_list");
                        $$->AddNode($1)->AddNode($2);
                        $$->row = $1->row;
                    }
                    ;

type_qualifier : CONST {
                    $$ = new TreeNode("type_qualifier_const");
                    $$->AddNode($1);
                    $$->row = $1->row;
                }
               | VOLATILE {
                    $$ = new TreeNode("type_qualifier_volatile");
                    $$->AddNode($1);
                    $$->row = $1->row;
               }
               ;

function_definition : type_specifier declarator '('
parameter_list_opt ')' compound_statement {
                        $$ = new TreeNode("function_definition");
                        $$->AddNode($1)->AddNode($2)->AddNode($4)-
>AddNode($6);
                        $$->row = $1->row;
                    }
                    ;
```

```
parameter_list_opt : parameter_list {
                        $$ = new TreeNode("parameter_list_opt");
                        $$->AddNode($1);
                        $$->row = $1->row;
                    }
                   | {
                        $$ = new TreeNode("parameter_list_opt");
                   }
                   ;

parameter_list : parameter_declaration {
                    $$ = new TreeNode("parameter_list");
                    $$->AddNode($1);
                    $$->row = $1->row;

                }
               | parameter_list ',' parameter_declaration {
                    $$ = new TreeNode("parameter_list");
                    $$->AddNode($1)->AddNode($3);
                    $$->row = $1->row;
               }
               ;

parameter_declaration : type_specifier declarator {
                            $$ = new
TreeNode("parameter_declaration");
                            $$->AddNode($1)->AddNode($2);
                            $$->row = $1->row;
                        }
                        | type_specifier declarator PURE_ASSIGN
initializer {
                            $$ = new TreeNode("init_declarator");
                            $$->AddNode($1)->AddNode($2)-
>AddNode($3)->AddNode($4);
                            $$->row = $1->row;
                        }


                      ;

initializer : assignment_expression {
                    $$ = new TreeNode("initializer");
                    $$->AddNode($1);
                    $$->row = $1->row;
```

```
                  }
              ;

compound_statement : '{' '}' {
                        $$ = new TreeNode("compound_statement");
                    }
                   | '{' statement_list '}' {
                        $$ = new TreeNode("compound_statement");
                        $$->AddNode($2);
                        $$->row = $2->row;
                   }
                   ;

statement_list : statement {
                    $$ = new TreeNode("statement_list");
                    $$->AddNode($1);
                    $$->row = $1->row;
                }
               | statement_list statement {
                    $$ = new TreeNode("statement_list");
                    $$->AddNode($1)->AddNode($2);
                    $$->row = $1->row;

               }
               ;

statement : expression_statement {
    $$ = new TreeNode("statement_exp");
    $$->AddNode($1);
    $$->row = $1->row;
}
| declaration {
    $$ = new TreeNode("statement_dec");
    $$->AddNode($1);
    $$->row = $1->row;
}
| compound_statement {
    $$ = new TreeNode("statement_comp");
    $$->AddNode($1);
    $$->row = $1->row;
}
| selection_statement {
    $$ = new TreeNode("statement_sel");
    $$->AddNode($1);
    $$->row = $1->row;
}
| iteration_statement {
    $$ = new TreeNode("statement_iter");
    $$->AddNode($1);
    $$->row = $1->row;
}
| jump_statement {
    $$ = new TreeNode("statement_jump");
    $$->AddNode($1);
    $$->row = $1->row;
}
| return_statement {
    $$ = new TreeNode("statement_return");
    $$->AddNode($1);
    $$->row = $1->row;
}
;

return_statement : RETURN expression_statement {
    $$ = new TreeNode("return_statement");
    $$->AddNode($1)->AddNode($2);
    $$->row = $1->row;
}
;

expression_statement : ';' {
    $$ = new TreeNode("expression_statement");
}
| assignment_expression ';' {
    $$ = new TreeNode("expression_statement");
    $$->AddNode($1);
    $$->row = $1->row;
}
;


assignment_expression : conditional_expression {
    // $$ = new TreeNode("assignment_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
```

```
}
| unary_expression assignment_operator assignment_expression {
    $$ = new TreeNode("assignment_expression");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
;

conditional_expression : logical_or_expression {
    // $$ = new TreeNode("conditional_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| logical_or_expression '?' assignment_expression ':'
conditional_expression {
    $$ = new TreeNode("conditional_expression");
    $$->AddNode($1)->AddNode($3)->AddNode($5);
    $$->row = $1->row;
}
;

logical_or_expression : logical_and_expression {
    // $$ = new TreeNode("logical_or_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| logical_or_expression OR logical_and_expression {
    $$ = new TreeNode("logical_or_expression");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
;

logical_and_expression : inclusive_or_expression {
    // $$ = new TreeNode("logical_and_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| logical_and_expression AND inclusive_or_expression {
    $$ = new TreeNode("logical_and_expression");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
;

inclusive_or_expression : exclusive_or_expression {
    // $$ = new TreeNode("inclusive_or_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| inclusive_or_expression ORBIT exclusive_or_expression {
    $$ = new TreeNode("inclusive_or_expression");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
;

exclusive_or_expression : and_expression {
    // $$ = new TreeNode("exclusive_or_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| exclusive_or_expression '^' and_expression {
    $$ = new TreeNode("exclusive_or_expression");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
;

and_expression : equality_expression {
    // $$ = new TreeNode("and_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| and_expression '&' equality_expression {
    $$ = new TreeNode("and_expression");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
;
```

```
equality_expression : relational_expression {
    // $$ = new TreeNode("equality_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| equality_expression EQUAL relational_expression {
    $$ = new TreeNode("equality_expression_eq");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
| equality_expression NOTEQUAL relational_expression {
    $$ = new TreeNode("equality_expression_ne");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
;

relational_expression : shift_expression {
    // $$ = new TreeNode("relational_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| relational_expression LESSTHAN shift_expression {
    $$ = new TreeNode("relational_expression_lt");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
| relational_expression GREATTHAN shift_expression {
    $$ = new TreeNode("relational_expression_gt");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
| relational_expression LESSEQUAL shift_expression {
    $$ = new TreeNode("relational_expression_le");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
| relational_expression GREATEQUAL shift_expression {
    $$ = new TreeNode("relational_expression_ge");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
;

shift_expression : additive_expression {
    // $$ = new TreeNode("shift_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| shift_expression LEFT_SHIFT additive_expression {
    $$ = new TreeNode("shift_expression_left");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
| shift_expression RIGHT_SHIFT additive_expression {
    $$ = new TreeNode("shift_expression_right");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
;

additive_expression : multiplicative_expression {
    // $$ = new TreeNode("additive_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| additive_expression '+' multiplicative_expression {
    $$ = new TreeNode("additive_expression_add");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
| additive_expression '-' multiplicative_expression {
    $$ = new TreeNode("additive_expression_sub");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
;

multiplicative_expression : cast_expression {
    // $$ = new TreeNode("multiplicative_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| multiplicative_expression '*' cast_expression {
    $$ = new TreeNode("multiplicative_expression_mul");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
| multiplicative_expression '/' cast_expression {
    $$ = new TreeNode("multiplicative_expression_div");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
;

cast_expression : unary_expression {
    // $$ = new TreeNode("cast_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
|  '(' type_specifier ')' cast_expression {
    $$ = new TreeNode("cast_expression");
    $$->AddNode($2)->AddNode($4);
    $$->row = $4->row;
}
;

unary_expression : postfix_expression {
    // $$ = new TreeNode("unary_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
       $$ = $1;
}
| increment unary_expression {
    $$ = new TreeNode("unary_expression_inc");
    $$->AddNode($1)->AddNode($2);
    $$->row = $1->row;
}
| decrement unary_expression {
    $$ = new TreeNode("unary_expression_dec");
    $$->AddNode($1)->AddNode($2);
    $$->row = $1->row;
```

```
}
| unary_operator cast_expression {
    $$ = new TreeNode("unary_expression_unary");
    $$->AddNode($1)->AddNode($2);
    $$->row = $1->row;
}
;

postfix_expression : primary_expression {
    // $$ = new TreeNode("postfix_expression");
    // $$->AddNode($1);
    // $$->row = $1->row;
    $$ = $1;
}
| postfix_expression '[' assignment_expression ']' {
    $$ = new TreeNode("postfix_expression_array");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
| postfix_expression '(' argument_expression_list_opt ')' {
    $$ = new TreeNode("postfix_expression_call");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
| postfix_expression '.' IDENTIFIER {
    $$ = new TreeNode("postfix_expression_struct");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
| postfix_expression arrow IDENTIFIER {
    $$ = new TreeNode("postfix_expression_arrow");
    $$->AddNode($1)->AddNode($2)->AddNode($3);
    $$->row = $1->row;
}
| postfix_expression increment {
    $$ = new TreeNode("postfix_expression_inc");
    $$->AddNode($1)->AddNode($2);
    $$->row = $1->row;
}
| postfix_expression decrement {
    $$ = new TreeNode("postfix_expression_dec");
    $$->AddNode($1)->AddNode($2);
    $$->row = $1->row;
```

```
}
;

argument_expression_list_opt : argument_expression_list {
    $$ = new TreeNode("argument_expression_list_opt");
    $$->AddNode($1);
    $$->row = $1->row;
}
|{
    $$ = new TreeNode("argument_expression_list_opt");
}
;

argument_expression_list : assignment_expression {
    $$ = new TreeNode("argument_expression_list");
    $$->AddNode($1);
    $$->row = $1->row;

}
| argument_expression_list ',' assignment_expression {
    $$ = new TreeNode("argument_expression_list");
    $$->AddNode($1)->AddNode($3);
    $$->row = $1->row;
}
;

primary_expression : IDENTIFIER {
    $$ = new TreeNode("primary_expression_id");
    $$->AddNode($1);
    $$->row = $1->row;
}
| INTNUM {
    $$ = new TreeNode("primary_expression_constantInt");
    $$->AddNode($1);
    $$->row = $1->row;
}
| FLOATNUM {
    $$ = new TreeNode("primary_expression_constantFloat");
    $$->AddNode($1);
    $$->row = $1->row;
}
| CHARACTERS {
    $$ = new TreeNode("primary_expression_constantChar");
    $$->AddNode($1);
    $$->row = $1->row;
}
| '(' assignment_expression ')' {
    $$ = new TreeNode("primary_expression_brace");
    $$->AddNode($2);
    $$->row = $2->row;
}
;

selection_statement : IF '(' assignment_expression ')' statement
%prec LOWER_THAN_ELSE {
    $$ = new TreeNode("selection_statement");
    $$->AddNode($1)->AddNode($3)->AddNode($5);
    $$->row = $1->row;
}
| IF '(' assignment_expression ')' statement ELSE statement {
    $$ = new TreeNode("iteration_statement");
    $$->AddNode($1)->AddNode($3)->AddNode($5)->AddNode($6)-
>AddNode($7);
    $$->row = $1->row;
}
// | SWITCH '(' assignment_expression ')' statement
;


iteration_statement : WHILE '(' assignment_expression ')' statement
{
    $$ = new TreeNode("iteration_statement");
    $$->AddNode($1)->AddNode($3)->AddNode($5);
    $$->row = $1->row;
}
| DO statement WHILE '(' assignment_expression ')' ';' {
    $$ = new TreeNode("iteration_statement");
    $$->AddNode($1)->AddNode($2)->AddNode($3)->AddNode($5);
    $$->row = $1->row;
}
| FOR '(' expression_opt ';' expression_opt ';' expression_opt ')'
statement {
    $$ = new TreeNode("iteration_statement");
    $$->AddNode($1)->AddNode($3)->AddNode($5)->AddNode($7)-
>AddNode($9);
    $$->row = $1->row;
```

```
}
| FOR '(' type_specifier init_declarator_list ';' expression_opt
';' expression_opt ')' statement {
    $$ = new TreeNode("iteration_statement");
    $$->AddNode($1)->AddNode($3)->AddNode($4)->AddNode($6)-
>AddNode($8)->AddNode($10);
    $$->row = $1->row;
}
;


expression_opt : assignment_expression {
    $$ = new TreeNode("expression_opt");
    $$->AddNode($1);
    $$->row = $1->row;
}
| {
    $$ = new TreeNode("expression_opt");
}
;

jump_statement : BREAK ';' {
    $$ = new TreeNode("jump_operator_break");
    $$->AddNode($1);
    $$->row = $1->row;
}
| CONTINUE ';' {
    $$ = new TreeNode("jump_operator_continue");
    $$->AddNode($1);
    $$->row = $1->row;
}
;

assignment_operator : PURE_ASSIGN {
    $$ = new TreeNode("assignment_operator_assign");
    $$->AddNode($1);
    $$->row = $1->row;
}
| ADD_ASSIGN {
    $$ = new TreeNode("assignment_operator_add");
    $$->AddNode($1);
    $$->row = $1->row;
}
```

```
| SUB_ASSIGN {
    $$ = new TreeNode("assignment_operator_sub");
    $$->AddNode($1);
    $$->row = $1->row;
}
| MUL_ASSIGN {
    $$ = new TreeNode("assignment_operator_mul");
    $$->AddNode($1);
    $$->row = $1->row;
}
| DIV_ASSIGN {
    $$ = new TreeNode("assignment_operator_div");
    $$->AddNode($1);
    $$->row = $1->row;
}
| MOD_ASSIGN {
    $$ = new TreeNode("assignment_operator_mod");
    $$->AddNode($1);
    $$->row = $1->row;
}
| LEFT_ASSIGN {
    $$ = new TreeNode("assignment_operator_left");
    $$->AddNode($1);
    $$->row = $1->row;
}
| RIGHT_ASSIGN {
    $$ = new TreeNode("assignment_operator_right");
    $$->AddNode($1);
    $$->row = $1->row;
}
| AND_ASSIGN {
    $$ = new TreeNode("assignment_operator_and");
    $$->AddNode($1);
    $$->row = $1->row;
}
| XOR_ASSIGN {
    $$ = new TreeNode("assignment_operator_xor");
    $$->AddNode($1);
    $$->row = $1->row;
}
| OR_ASSIGN {
    $$ = new TreeNode("assignment_operator_or");
    $$->AddNode($1);
    $$->row = $1->row;
}
;

unary_operator : '&'{
    $$ = new TreeNode("unary_operator_addr");
    // $$->row = $1->row;
}
| '*'{
    $$ = new TreeNode("unary_operator_ptr");

    // $$->row = $1->row;
}
| '+'{
    $$ = new TreeNode("unary_operator_pos");

    // $$->row = $1->row;
}
| '-'{
    $$ = new TreeNode("unary_operator_neg");

    // $$->row = $1->row;
}
| '~'{
    $$ = new TreeNode("unary_operator_bitneg");

    // $$->row = $1->row;
}
| '!'{
    $$ = new TreeNode("unary_operator_not");

    // $$->row = $1->row;
};
```

3.3 打印语法树

打印语法树会按照先序遍历的方式，打印出语法树每一节点的代码行数、节点类型、节点内容。

```
void ShowSyntaxTree(SyntaxTree cur, int depth)
{
    string pre = "|";
    for (int i = 0; i < depth; i++)
    {
        pre += '-';
    }

    if (cur->type == "IDENTIFIER")
        cout << pre << "ID(" << cur->value << ")";
    else if (cur->type == "INTNUM")
        cout << pre << "NUM(" << cur->value << ")";
    else
        cout << pre << cur->value;

    if(cur->attr.type == Int)
    {
        cout << " --row=" << cur->row;
        cout << "; attr=INT" << endl;
    }
    else if(cur->attr.type == Array)
    {
        cout << " --row=" << cur->row;
        cout << "; attr=Arsray" << endl;
    }
    else
    {
        if(cur->row != 0)
            cout << " --row=" << cur->row;
        cout << endl;
    }

    if (!cur->child.empty())
    {
        auto next = cur->child.begin();
        for (; next < cur->child.end(); next++)
            ShowSyntaxTree(*next, depth + 1);
    }
}
```

3.4 CFG报错信息

在接收到CFG没有定义的规则时，调用yyerror抛出错误。

```
void yyerror(char* str) {
    fprintf(stderr, "%s", str);
}
```

3.5 主函数入口

语法分析的主函数入口，包括符号表创建、打印语法树、语义分析、生成并打印IR、打印汇编代码。

```
int main()
{
    SymbolTable T = SymbolTable();
    ofstream ofs;
    ofs.open("SyntaxTree.txt", ios::out);
    yydebug = 1;
    yyparse();
    cout<<"======== SyntaxTree ========"<<endl;
    ShowSyntaxTree(root, 0);

    cout<<"Start Parsing SyntaxTree..."<<endl;
    T.create_table();
    parse(root);
    if(!errinfo.empty())
        ShowErr(errinfo);
    cout<<"Finished."<<endl;

    cout<<endl<<"========================== Annotated SyntaxTree
========================="<<endl;
    ShowSyntaxTree(root, 0);
    SaveSyntaxTree(root, 0, ofs);
    ofs.close();
    if(errinfo.empty())
    {
        GenerateIR(root);
        cout<<endl<<"====================== Intermediate
Representation ======================"<<endl;

        IRofs.open("pseudo_asm.s", ios::out);
        ofs.open("TargetCode.s", ios::out);
        targetCodes.asm_head();
        targetCodes.asm_io();
        IRCodes.printIR();
        ofs.close();
        IRofs.close();
        cout<<endl<<"====================== Target Code 
Representation ======================"<<endl;

        ifs.open("TargetCode.s", ios::in);
        targetCodes.printTarget();
        ifs.close();
    }
}
```

4 语义分析

5 生成中间表示语言

5.1 三地址码格式

中间表示语言以三地址码的形式储存，对于本编译器中使用的三地址码语法规定为（本语法描述以int类型示例，其余数据类型同理；运算符以加法示例，其余运算符同理；循环语句以for示例，其余语句同理）：

三地址码语法	语法描述
DEC INT a	声明int型变量a
DEC INT a=1	声明int型变量a，初始值为1
DEC INT a[10]	声明长度为10的int型数组a
DEC INT a[10]=0	声明长度为10的int型数组a，初始值为0
DEC INT *a	声明int型指针a
FUNC INT f	声明函数f()，返回值为int型
PARAM INT a	声明函数的形式参数int型变量a
PARAM INT a=1	声明函数的形式参数int型变量a，缺省值为1
NO PARAM	函数没有参数

三地址码语法	语法描述
BODY	函数体的开始标志
a=1	将1赋值给变量a（含变量类型转换）
temp_1=1+2	将1+2的结果赋值给temp_1（编译器生成的临时变量）
CALL f	调用函数f
ARG 0	调用函数时传递参数0（按序传递）
NO ARGUMRNT	调用函数时不传递参数
ARGUMENT END	调用函数传递参数结束
IF temp_2	if语句（temp_2储存条件语句的结果）
LABEL label_1	放置label_1
GOTO label_1	跳转至label_1
RET 0	返回0
f END	函数体结束
a = (int)b	将1强制类型转换为int类型并赋值给a

5.2 三地址码转换规则

三地址码最多只能储存3个符号，所以对于需要储存的信息超过3的语句进行重构：

从优先级的角度出发，将语句分割成多条信息长度不多于3的语句序列，创建临时变量来储存过程中产生的值，如

```
a = (1+2)*3
```

会被处理为

```
temp_1 = 1+2
a = temp_1*3
```

又比如

```
if(a<0){
  a = 1;
}
```

会被处理为

```
temp_1 = a<0
if temp_1 GOTO label_1
LBAEL label_1
a=1
```

从而将所有语句以三地址码的格式储存。

5.3 三地址码中元素数据结构

三地址码的每个地址数据结构包括节点的类型、地址的名称和地址的值。

```
enum AddrKind { VARIABLE, ARRAY, CONSTANT, STRING, EMPTY,
VARIABLEINIT, ARRAYINIT, POINT, OP, BRACEEXP, POSTFIX };

class Addr_ {
    static int temp;

    public:
        AddrKind kind; //节点类型
        string name; //地址名称
        int value;   //地址值
};
```

从实际处理来说，节点的类型仅为标注与区分作用，故在创建节点的时候可以没有节点的类型。对于不同类型的节点，名称和值都不总是被需要，故构造函数为

```
        Addr_(AddrKind kind, string name, int value) {
            this->kind = kind;
            this->name = name;
            this->value = value;
            // cout<<"type1:"<<kind<<name<<value<<endl;
        }
        Addr_(string name, int value) {
            this->name = name;
            this->value = value;
            // cout<<"type1:"<<kind<<name<<value<<endl;
        }
        Addr_(string name) {
            this->name = name;
            // cout<<"type1:"<<kind<<name<<value<<endl;
        }
        Addr_(AddrKind kind, string name) {
            this->kind = kind;
            this->name = name;
            this->value = 0;
        }
        Addr_(AddrKind kind, int value) {
            this->kind = kind;
            this->value = value;
            this->name = "_none_";
            // cout<<"type2:"<<kind<<endl;

        }
        Addr_(AddrKind kind) {
            this->kind = kind;
            this->name = "_none_";
            this->value = 0;
            // cout<<"type3:"<<kind<<endl;
        }

        Addr_() {
            this->kind = VARIABLE;
            this->name = "temp_" + to_string((this->temp)++);
            this->value = 0;
        }
```

如对于语句a=1，生成3个地址节点，分别为：

```
Addr a {
kind = VARIABLE;
name = "a";
}

Addr op{
kind = OP;
name = "EQUAL";
}

Addr value{
kind = NUM;
value = 1;
}
```

5.4 三地址码数据结构

中间表示语言的每个节点实际上是一个链表的节点，包括节点的类型、三地址码。

```
class InterCode {

    public:
        OpKind kind;
        Addr addr1, addr2, addr3;
        InterCode() {}
};
```

如语句 a = 1，此处形成节点（没有赋值的属性在后续处理时也不会被使用）

```
InterCode a,
a->kind = ASSIGNMENT,
a->addr1->name = a,
a->addr2->name = 1
```

5.5 中间表示语言数据结构

中间表示语言的数据结构实际上是由三地址码连接形成的链表。

整个程序只会生成一个IRCode类，其中包括一个IR_list，程序的代码倒序排列在链表上（代码的顺序不会影响IR和RISC-V汇编代码的生成，只会在打印的时候需要进行尾递归）。

```
class IRCode {
    IRList IR_list;
}
```

5.6 生成中间表示语言

5.6.1 主函数入口

对于中间表示语言的生成，从最高层分为函数声明、变量声明、语句（包括函数体等）。在遍历语法树的时候，标记已经遍历过的节点，整体上通过从上到下（root到leaf）、从左到右的顺序遍历。

特殊地，此处只会处理对全局变量的声明，而函数内部的局部变量声明则会在statement_list处理的过程中调用函数来生成中间表示代码；

特殊地，此处处理的statement_list不包括函数体，函数体的处理需要在function_definition处理后调用GenerateIR进行处理（函数的声明必须伴随定义），即在最开始的时候不会被触发。

```
void GenerateIR(SyntaxTree cur) {
    //对于数组，需要申请内存空间
    // cout << "IR"<<cur->value << endl;

    if (cur->value == "function_definition"){
        generate_func(cur);
        // GenerateIR(cur->child[3]);
        return;
    }
    else if (cur->value == "declaration"){
        // 在此进入的一定是全局变量
        generate_val(cur);
        return;
    }
    else if (cur->value == "statement_list"){
        generate_statement_list(cur);
        cur->attr.isdone = 1;
        return;
    }

    if (!cur->child.empty()) {
        // 遍历子节点
        auto next = cur->child.begin();
        for (; next < cur->child.end(); next++) {
            GenerateIR(*next);
        }
    }
}
```

5.6.2 函数定义

函数定义的处理实现函数类型、函数名、参数列表的中间表示代码生成，同时调用主函数分析函数体（函数声明对最后的汇编代码没有意义，故没有实现）。

```
void generate_func(TreeNode *cur) {  //包括函数指针和普通函数
    if(cur->child[1]->child.size() == 1){  //ID(declartor)
    // cout<<"function_definition->declarator(1)-
>direct_declarator"<<endl;
        Addr dest = new Addr_(STRING, cur->child[1]->child[0]-
>child[0]->value);
        Addr typedest = new Addr_(STRING,cur->child[0]->child[0]-
>value);
        InterCode code(OpKind::FUNC, typedest, dest);
        IRCodes.insert(code);
        generate_params(cur->child[2]);  //parameter_list_opt
        GenerateIR(cur->child[3]);
        cout<<"111"<<endl;
        InterCode code2(OpKind::FUNCEND, dest);
        IRCodes.insert(code2);
    }
    else{                                  //ID(pointer declartor)
    cout<<"function_definition->declarator(1)->pointer(0)
direct_declarator"<<endl;
        Addr dest = new Addr_(STRING, "*"+cur->child[1]->child[0]-
>child[1]->child[0]->value+" "+cur->child[1]->value);
        InterCode code(OpKind::FUNC, dest);
        IRCodes.insert(code);
        generate_params(cur->child[2]);  //parameter_list_opt
        GenerateIR(cur->child[3]);
    }
}
```

在CFG中，函数声明的推导过程为（以函数int f(char b)为例）：

```
function_definition->type_specifier declarator '('
parameter_list_opt ')' compound_statement //int f(char b)
type_specifier->INT   //int
declarator>direct_declarator->IDENTIFIER //f
parameter_list_opt->parameter_list->parameter_declaration-
>type_specifier declarator //char b
type_specifier->CHAR  //char
declarator->direct_declarator->IDENTIFIER  //b
```

对应的函数调用序列为：

```
generate_func(TreeNode *cur)  //函数声明
->generate_params(cur->child[2])  //参数列表
->GenerateIR(cur->child[3])  //函数体
```

在生成IR的同时，生成汇编代码。如此处的函数调用序列为（函数体生成的汇编代码需要每条具体语句调用不同的处理函数）：

```
targetCodes.asm_function(str_addr(node->code.addr1))->
          targetCodes.asm_function_argument(str_addr(node-
>code.addr1))   //函数声明->参数列表
```

5.6.3 变量声明

变量声明生成变量的类型、变量的标识符，在必要的时候可以生成变量的初始值。

```
Addr generate_val_list(string type,TreeNode *cur) {
    if(cur->child.size() == 2){
    //    cout<<"init_declarator_list->init_declarator_list ','
init_declarator"<<endl;
       generate_val_list(type,cur->child[0]);
       generate_val_declarator(type,cur->child[1]);
    }
    else{
    //    cout<<"init_declarator_list->init_declarator"<<endl;
       generate_val_declarator(type,cur->child[0]);
    }
}
```

```
Addr generate_val(TreeNode *cur){
    //cout<<"declaration->type_specifier init_declarator_list"
<<endl;
    return generate_val_list(cur->child[0]->child[0]->value,cur-
>child[1]);
}
```

在CFG中，变量声明的推导过程为（以int a = 1为例）：

```
declaration->type_specifier init_declarator_list  //int a = 1
type_specifier->INT   //int
init_declarator_list->init_declarator->declarator PURE_ASSIGN
initializer  //a=1
declarator>direct_declarator->IDENTIFIER  //a
PURE_ASSIGN->EQUAL   //=
initializer->assignment->INTNUM  //1
```

对应的函数调用序列为：

```
generate_val(TreeNode *cur)  //声明变量
->generate_val_list(string type,TreeNode *cur)  //变量列表
->generate_val_declarator(string type,TreeNode *cur)  //变量名
->generate_assign_declarator(TreeNode *cur1,TreeNode *cur2,string
type)  //变量初始化
```

生成汇编代码调用的函数序列为：

```
targetCodes.asm_dec_variable((str_addr(node-
>code.addr1),str_addr(node->code.addr2),str_addr(node->code.addr3))
//变量声明
```

5.6.4 语句

对于语句，我们有以下的分类：

语句	描述	调用函数
表达式（expression）	一般表达式	generate_exp
声明（declaration）	声明变量（不包括函数）	generate_val

语句	描述	调用函数
主体语句（compond）	if、else、while、for、函数的主体	generate_statement
选择语句（selection）	if、else	generate_if_stmt
循环语句（iteration）	while、for、do-while	generate_iter
返回语句（return）	return	generate_ret_stmt
跳转语句（jump）	break，continue	generate_jump

在每个处理函数内部，依据CFG的推导规则，循环调用相应的处理函数直至到达终结符。

a.表达式语句

对于表达式语句的处理，由于在生成语法树的时候，对于不生成更多数量子节点的推导规则我们的处理是直接合并子节点和该节点，所以在推导后一定会直接落在某个终结符上。对于有多个子节点的节点，按照语句的执行顺序生成对应IR并调用函数生成汇编。所以，只要是由assignment_expression推导出的规则都可以通过调用generate_exp函数本身来得到。

实际上，对于某些分支可以进行合并，但为了使代码的结构更加直观且符合CFG的规则，此处并没有进行合并。

```
Addr generate_exp(TreeNode *cur){
    Addr a1, a2, a3;
    if(cur->value == "assignment_expression"){
        cout<<"assign"<<endl;
        a1 = generate_lv(cur->child[0]);
        a2 = generate_assignment_operator(cur->child[1]);
        a3 = generate_exp(cur->child[2]);
        //cout<<a1->name<<"  "<<a3->value<<endl;
        InterCode code(ASSIGNMENT, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "conditional_expression"){
        //toDO
        a1 = generate_exp(cur->child[0]);
        a2 = generate_exp(cur->child[1]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(ASSIGNMENT, a1, a2, a3);
        IRCodes.insert(code);
        return a2;
    }
    else if(cur->value == "logical_or_expression"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(LOGICOR, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "logical_and_expression"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(LOGICAND, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "inclusive_or_expression"){
        a1 = new Addr_();
        // cout<<"BITOR"<<endl;
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(BITOR, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "exclusive_or_expression"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(BITXOR, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "and_expression"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(BITAND, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "equality_expression_eq"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(EQ, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }else if(cur->value == "equality_expression_ne"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(NE, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "relational_expression_lt"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(LT, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "relational_expression_gt"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(GT, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "relational_expression_le"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(LE, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "relational_expression_ge"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(GE, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "shift_expression_left"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(LEFT, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "shift_expression_right"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[2]);
        InterCode code(RIGHT, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "additive_expression_add"){
        cout<<"add"<<endl;
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(ADD, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "additive_expression_sub"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(SUB, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "multiplicative_expression_mul"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(MUL, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "multiplicative_expression_div"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(DIV, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "cast_expression"){
        a1 = new Addr_();
        cout<<"111"<<endl;
        a2 = generate_exp(cur->child[1]);
        enum OpKind ty;
        cout<<cur->child[0]->child[0]->value<<endl;

        if(cur->child[0]->child[0]->value=="INT")ty=TOINT;
        if(cur->child[0]->child[0]->value=="CHAR")ty=TOCHAR;
        if(cur->child[0]->child[0]->value=="FLOAT")ty=TOFLOAT;
        InterCode code(ty,a1,a2);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "unary_expression_inc"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[1]);
        InterCode code(PREINC,a1,a2);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "unary_expression_dec"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[1]);
        InterCode code(PREDEC,a1,a2);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "unary_expression_unary"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[1]);
        enum OpKind ty;
        if(cur->child[0]->value=="unary_operator_addr")ty=GETADDR;
        if(cur->child[0]->value=="unary_operator_ptr")ty=PTR;
        if(cur->child[0]->value=="unary_operator_pos")ty=POS;
        if(cur->child[0]->value=="unary_operator_neg")ty=NEG;
        if(cur->child[0]->value=="unary_operator_bitneg")ty=BITNEG;
        if(cur->child[0]->value=="unary_operator_not")ty=NOT;
        InterCode code(ty,a1,a2);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "postfix_expression_array"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        a3 = generate_exp(cur->child[1]);
        InterCode code(VISITARRAY, a1, a2, a3);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "postfix_expression_call"){
        cout<<"callfunc"<<endl;
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        cout<<"call"<<endl;
        InterCode code(CALL,a1,a2);
        IRCodes.insert(code);
        generate_args(cur->child[1]);
        cout<<"callfinish"<<endl;
        return a1;
    }
    else if(cur->value == "postfix_expression_struct"){

    }
    else if(cur->value == "postfix_expression_arrow"){

    }
    else if(cur->value == "postfix_expression_inc"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        InterCode code(POSTINC,a1,a2);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "postfix_expression_dec"){
        a1 = new Addr_();
        a2 = generate_exp(cur->child[0]);
        InterCode code(POSTDEC,a1,a2);
        IRCodes.insert(code);
        return a1;
    }
    else if(cur->value == "primary_expression_id"){
        a1 = new Addr_(cur->child[0]->value);
        return a1;
    }
    else if(cur->value == "primary_expression_constantInt"){
        a1 = new Addr_(cur->child[0]->value, cur->child[0]-
>attr.numval);
        return a1;
    }
    else if(cur->value == "primary_expression_constantFloat"){
        a1 = new Addr_(cur->child[0]->value, cur->child[0]-
>attr.fnumval);
        return a1;
    }
    else if(cur->value == "primary_expression_constantChar"){
        a1 = new Addr_(cur->child[0]->value, cur->child[0]-
>attr.cnumval);
        return a1;
    }
    else if(cur->value == "primary_expression_brace"){
        a1 = generate_exp(cur->child[0]);
        return a1;
    }
    else if (cur->value == "expression_opt"){
        a1 = generate_exp(cur->child[0]);
        return a1;
    }
}
```

b.声明语句

对于声明语句的处理与5.6.3节的变量声明处理是一致的，因为在IR部分没有对全局变量和局部变量进行区别，而是在汇编层进行区别（储存的位置和释放寄存器的时间点）。

c.主体语句

对于主体语句的处理与5.6.4节的statement_list处理是一致的，因为它们本质上都是代码块，区别只是在于位置和执行的时间点，如果代码块内发生运行时错误，在IR层面也不会进行处理，而是在汇编层进行抛出。

d.选择语句

对于选择语句，实际上只会出现两种情况：if-else或者if，可以通过子节点的数量进行区别。

在处理选择语句的时候，通过临时变量（temp）来储存判断条件的结果，通过标签（LABEL）来标记条件成立或者错误时的跳转（GOTO）地址。

```
Addr generate_if_stmt(TreeNode *cur){
    if(cur->child.size() == 3){
        //cout<<"if then"<<endl;
        Addr label1 = new Addr_(AddrKind::LAB, new_label());
        Addr label2 = new Addr_(AddrKind::LAB, new_label());
        Addr ass = generate_exp(cur->child[1]);
        InterCode code1(SEL,ass,label1);
        InterCode code2(GOTO,label2);
        InterCode code3(LABEL,label1);
        InterCode code4(LABEL,label2);
        IRCodes.insert(code1);
        IRCodes.insert(code2);
        IRCodes.insert(code3);
        Addr state = generate_statement(cur->child[2]);
        IRCodes.insert(code4);
    }
    if(cur->child.size() == 5){
        // LABEL label1 = new_label();
        Addr ass = generate_exp(cur->child[1]);
        Addr label1 = new Addr_(AddrKind::LAB, new_label());
        Addr label2 = new Addr_(AddrKind::LAB, new_label());
        InterCode code1(SEL,ass,label1);
        InterCode code2(GOTO, label2);
        InterCode code3(LABEL,label1);
        InterCode code4(LABEL,label2);
        IRCodes.insert(code1);
        Addr elsestmt = generate_statement(cur->child[4]);
        IRCodes.insert(code2);
        IRCodes.insert(code3);
        Addr state = generate_statement(cur->child[2]);
        IRCodes.insert(code4);
    }
}
```

e.循环语句

循环语句包括for、while、do-while，同时for的第一条语句可能是变量声明，所以分成四种情况进行讨论。

对于循环的条件，使用临时变量来保存其bool值；在循环体和出口打上标签（LABEL），根据循环条件的bool值进行对应的跳转（GOTO）。

同时，只允许for的第一条语句出现变量声明，其余情况"( )"中不允许出现变量声明。

```
Addr generate_iter(TreeNode *cur){
    Addr addr1, addr2, addr3;
    if(cur->child.size()==3){
        // while
        Addr label1 = new Addr_(AddrKind::LAB, new_label());
        Addr label2 = new Addr_(AddrKind::LAB, new_label());
        Addr label3 = new Addr_(AddrKind::LAB, new_label());
        InterCode goto1Code(GOTO, label1);
        InterCode goto3Code(GOTO, label3);
        InterCode label1Code(LABEL, label1);
        InterCode label2Code(LABEL, label2);
        InterCode label3Code(LABEL, label3);

        IRCodes.insert(label1Code);                        // label 1:

        Addr ass = generate_exp(cur->child[1]);
        InterCode selCode(SEL, ass, label2);
        IRCodes.insert(selCode);                           // judge
goto label2
        IRCodes.insert(goto3Code);                         // goto
label3
        IRCodes.insert(label2Code);                        // label 2:
        generate_statement(cur->child[2]);                 // body
        IRCodes.insert(goto1Code);                       // goto
label 1
        IRCodes.insert(label3Code);                      // label 3:

    }else if(cur->child.size()==4){
        //Do while
        Addr label1 = new Addr_(AddrKind::LAB, new_label());
        Addr label2 = new Addr_(AddrKind::LAB, new_label());
        InterCode label1Code(LABEL, label1);
        IRCodes.insert(label1Code);                      // label 1:
        generate_statement(cur->child[1]);               // body
        Addr ass = generate_exp(cur->child[3]);
        InterCode selCode(SEL, ass, label1);
        IRCodes.insert(selCode);                         // judge
goto label 1

    }else if(cur->child.size()==5){
        //for without                    FOR (exp1; judge; exp2)
body
        generate_exp(cur->child[1]);                    //      exp1
        Addr label1 = new Addr_(AddrKind::LAB, new_label());
        Addr label2 = new Addr_(AddrKind::LAB, new_label());
        Addr label3 = new Addr_(AddrKind::LAB, new_label());
        InterCode goto1Code(GOTO, label1);
        InterCode goto2Code(GOTO, label2);
        InterCode goto3Code(GOTO, label3);
        InterCode label1Code(LABEL, label1);
        InterCode label2Code(LABEL, label2);
        InterCode label3Code(LABEL, label2);
        IRCodes.insert(label1Code);                      // label 1:
        Addr ass = generate_exp(cur->child[2]);
        InterCode selCode(SEL, ass, label1);
        IRCodes.insert(selCode);                         // judge
goto label 2
        IRCodes.insert(goto3Code);                       // goto
label 3
        IRCodes.insert(label2Code);                      // label 2:
        //cout<<"body "<<cur->child[4]->child.size()<<endl;
        generate_statement(cur->child[4]);               // body
        generate_exp(cur->child[3]);               // exp2
        IRCodes.insert(goto1Code);                       // goto
label 1
        IRCodes.insert(label3Code);                      // label 3:
    }else if(cur->child.size()==6){
        //for with declarator
        Addr label1 = new Addr_(AddrKind::LAB, new_label());
        Addr label2 = new Addr_(AddrKind::LAB, new_label());
        Addr label3 = new Addr_(AddrKind::LAB, new_label());
        InterCode goto1Code(GOTO, label1);
        InterCode goto2Code(GOTO, label2);
        InterCode goto3Code(GOTO, label3);
        InterCode label1Code(LABEL, label1);
        InterCode label2Code(LABEL, label2);
        InterCode label3Code(LABEL, label2);
        generate_val_list(cur->child[1]->child[0]->value, cur-
>child[2]);
        IRCodes.insert(label1Code);                        // label 1:
        Addr ass = generate_exp(cur->child[3]);
        InterCode selCode(SEL, ass, label1);
        IRCodes.insert(selCode);                           // judge
goto label 2
        IRCodes.insert(goto3Code);                         // goto
label 3
        IRCodes.insert(label2Code);                        // label 2:
        generate_statement_list(cur->child[5]);                 //
body
        generate_exp(cur->child[4]);                 // exp2
        IRCodes.insert(goto1Code);                         // goto
label 1
        IRCodes.insert(label3Code);                        // label 3:
    }
}
```

f.返回语句

对于返回语句，通过子节点的数量可以分为返回一个数据或者没有返回值。

```
Addr generate_ret_stmt(TreeNode *cur){
    cout<<cur->child.size()<<"   "<<cur->child[1]->value<<endl;
    if(cur->child[1]->child.size()==1){
        Addr dest = generate_exp(cur->child[1]->child[0]);
        InterCode code(RET, dest);
        // cout<<"ret:"<<dest->name<<endl;
        IRCodes.insert(code);
    }else{
        Addr dest = new Addr_(EMPTY);
        InterCode code(RET, dest);
        // cout<<"ret:"<<endl;
        IRCodes.insert(code);
    }
}
```

g.跳转语句

对于跳转语句的处理，实际上就是使用GOTO语句跳转到不同的标签（LABEL）位置。

对于break语句，实际上就是直接跳转到主体语句的出口；

对于continue语句，实际上就是直接跳转到循环语句的判断条件语句；