

实验报告：语义分析/编译器

涂奕腾 2020201018

一、词法分析

1. 行、列计数

行和列的计数主要使用定义在声明部分的计数器实现：每当匹配一个字符后(除\n,\t)应当加上字符长度 `yyleng`，对于\n 应当将列计数器 `col` 重置为 1、将行计数器+1；对于\t 应当对列计数器+8

```
%{
int row = 1, col = 1;
%}

\n          {++row; col = 1;}
\t          {col += 8;}
" "         {++col;}
```

2. 关键字、界符、算符

暴力匹配即可：

```
"if"      {col += yyleng; yylval = ++tot; return IF;}
"else"    {col += yyleng; yylval = ++tot; return ELSE;}
"while"   {col += yyleng; yylval = ++tot; return WHILE;}
"continue" {col += yyleng; yylval = ++tot; return CONTINUE;}
"break"   {col += yyleng; yylval = ++tot; return BREAK;}
"return"  {col += yyleng; yylval = ++tot; return RETURN;}
"const"   {col += yyleng; yylval = ++tot; return CONST;}
"int"     {col += yyleng; yylval = ++tot; return INT;}
"void"    {col += yyleng; yylval = ++tot; return VOID;}
"main"    {col += yyleng; yylval = ++tot; return MAIN;}

"+"      {col += yyleng; yylval = ++tot; return '+';}
"_"      {col += yyleng; yylval = ++tot; return '-'}
"*"      {col += yyleng; yylval = ++tot; return '*'}
"/"      {col += yyleng; yylval = ++tot; return '/'}
"%"      {col += yyleng; yylval = ++tot; return '%'}
"<"      {col += yyleng; yylval = ++tot; return '<'}
">"      {col += yyleng; yylval = ++tot; return '>'}
"!"      {col += yyleng; yylval = ++tot; return '!'}
"="      {col += yyleng; yylval = ++tot; return '='}
"<="     {col += yyleng; yylval = ++tot; return LEQ;}
">="     {col += yyleng; yylval = ++tot; return GEQ;}
"=="     {col += yyleng; yylval = ++tot; return EQ;}
```

```
"!=" {col += yyleng; yylval = ++tot; return UEQ;}
"&&" {col += yyleng; yylval = ++tot; return AND;}
"||" {col += yyleng; yylval = ++tot; return OR;}
```

3. 标识符

标识符应当满足变量/函数的命名规则：以下划线或字母开头，后面由下划线、字母、数字组成，用正则表达式直接匹配

```
IDENTIFIER [_a-zA-Z][_a-zA-Z0-9]*
{IDENTIFIER} {printf("%s:\t I, (%d, %d)\n",yytext,row,col); col += yyleng;}
```

4. 常数

八进制和十六进制的常数可以直接使用正则表达式判断，十进制则按照 `sysy` 语言的要求进行匹配。

另一方面，为了符合词法定义，数字后面应当只能接空格符(`SPACE`、`\n`、`\t`)、界符和操作符，即排除了非法符号、关键字和标识符，储存在 `afterNUMBER` 中

```
DECIMAL (-?)([1-9][0-9+])|([0-9])
OCTALCONS (-?)0[0-7]+
HEXCONS (-?)0[xX][0-9a-fA-F]+
NUMBER {OCTALCONS}|{HEXCONS}|{DECIMAL}
afterNUMBER {OPERATOR}|{BORDER}|" "|\\n\\t\\v\\r\\f
{NUMBER}/{afterNUMBER} {col += yyleng; yylval = atoi(yytext); strcpy(tag[tot],
yytext); return NUMBER;}
```

5. 其他字符

主要是针对出现非法字符(如`@`)以及常数后接标识符(如变量名)/关键字的情况

```
other [^ \n\t\v\r\f\(\)\{\}\[\]\;\,\\"'+-\\*\/%<>\\=\&\\|!\\^~]
{other}* {printf("%s:\t T, (%d, %d)\n",yytext,row,col); col += yyleng;}
```

6. 注释

使用条件激活规则判断注释：

对于单行注释，当检测到“`//`”后激活单行注释模式，实际上本行的内容都没有意义了，可以直接将行计数器 `row+1`，列计数器重设置为 `1`，将本行读完即可(即读到`\n`)

对于多行注释，当检测到“`/*`”后激活单行注释模式，行、列的计数和正常情况下相同，直到读到“`\n`”退出多行注释模式

```
notebegin "/*"
noteend "*/"
linenotebegin "//"
linenoteend \n
%x NOTE
%x LINENOTE
```

```
{notebegin}          {BEGIN NOTE; col += yyleng;}
<NOTE>\n             {++row; col = 1;}
<NOTE>\t             {col += 8;}
<NOTE>.              {col += yyleng;}
<NOTE>{noteend}       {BEGIN INITIAL; col += yyleng;}
{linenotebegin}       {BEGIN LINENOTE;}
<LINENOTE>.          {}
<LINENOTE>{linenoteend} {BEGIN INITIAL; ++row; col=1;}
```

7. 其他注意事项

在最后的语义分析中，由于最终的处理使用的时 `g++` 编译器，需要在 `.l` 文件开头加上下面一行代码以保证正确编译：

```
%option noyywrap
```

二、语法分析

1. 语法规则

尽量使用左递归对语法规则改造如下(部分语法规则在语义分析中还需要进行修改):

```
CompUnits: CompUnit | CompUnits CompUnit
CompUnit: Decl | FuncDef
Decl: ConstDecl | VarDecl
ConstDecls: ConstDef | ConstDecls ',' ConstDef
ConstDecl: CONST INT ConstDecls ';'
ConstDefs: '[' ConstExp ']' | ConstDefs '[' ConstExp ']'
ConstDef: ID '=' ConstInitVal | ID ConstDefs '=' ConstInitVal
ConstInitVals: ConstInitVal | ConstInitVals ',' ConstInitVal
ConstInitVal: ConstExp | '{' '}' | '{' ConstInitVals '}'
VarDecls: VarDef | VarDecls ',' VarDef
VarDecl: INT VarDecls ';'
VarDefs: '[' ConstExp ']' | VarDefs '[' ConstExp ']'
VarDef: ID | ID '=' InitVal | ID VarDefs | ID VarDefs '=' InitVal
InitVals: InitVal | InitVals ',' InitVal
InitVal: Exp | '{' '}' | '{' InitVals '}'
FuncDef: VOID ID '(' ')' Block | INT ID '(' ')' Block | VOID ID '(' FuncFParams ')' Block
| INT ID '(' FuncFParams ')' Block
FuncFParams: FuncFParam | FuncFParams ',' FuncFParam
FuncFParam: INT ID | INT ID '[' ']' | INT ID '[' ']' LVals
Blocks: BlockItem | Blocks BlockItem
Block: '{' '}' | '{' Blocks '}'
BlockItem: Decl | Stmt
Stmt: LVal '=' Exp ';' | Exp ';' | ';' | Block | IF '(' Cond ')' Stmt | IF '(' Cond ')'
| ELSE Stmt | WHILE '(' Cond ')' Stmt | BREAK ';' | CONTINUE ';' | RETURN Exp ';' |
| RETURN ';'
Exp: AddExp
Cond: LOrExp
LVals: '[' Exp ']' | LVals '[' Exp ']'
LVal: ID | ID LVals
PrimaryExp: '(' Exp ')' | LVal | NUMBER
UnaryExp: PrimaryExp | ID '(' ')' | ID '(' FuncRParams ')' | '+' UnaryExp | '-' UnaryExp
| '!' UnaryExp
FuncRParams: Exp | FuncRParams ',' Exp
MulExp: UnaryExp | MulExp '*' UnaryExp | MulExp '/' UnaryExp | MulExp '%' UnaryExp
AddExp: MulExp | AddExp '+' MulExp | AddExp '-' MulExp
RelExp: AddExp | RelExp '<' AddExp | RelExp '>' AddExp | RelExp LEQ AddExp | RelExp GEQ
AddExp
EqExp: RelExp | EqExp EQ RelExp | EqExp UEQ RelExp
LAndExp: EqExp | LAndExp AND EqExp
LOrExp: LAndExp | LOrExp OR LAndExp
ConstExp: AddExp
```

2. if-else 移进-规约冲突

通过%nonassoc 指定

```
%nonassoc WITHOUTELSE
%nonassoc ELSE
IF '(' Cond ')' Stmt %prec WITHOUTELSE
IF '(' Cond ')' Stmt ELSE Stmt
```

3. 记录分析过程

设计以下数据结构记录分析过程和生成语法树过程，**tot** 和 **num** 分别表示了两种不同的编号方式，**tot** 记录匹配时创建节点时的编号；**num** 为构造/输出语法树时各个节点的编号。**Tag** 记录某个节点的标签。**Node** 结构体记录了语法树的相关信息：某个节点有多少个子节点(**siz**)，子节点分别是哪些由 **tot** 记录(**ch**)，父节点由 **num** 记录(**fa**)，该节点是父节点的哪个儿子由 **num** 记录(**which_ch**)。结构定义与构造语法树如下：

```
int tot = 0; //number of tree node
int num = 0; //node number
char tag[114514][100];
typedef struct {int ch[15], siz, fa, which_ch; }Node;
Node node[114514];

void dfs(int cur){
    fprintf(f1, "node%d[label = \"", ++num);
    for(int i = 0; i < node[cur].siz; ++i){
        node[node[cur].ch[i]].fa = num, node[node[cur].ch[i]].which_ch = i;
        //输出子节点
    }
    //输出连边
    for(int i = 0; i < node[cur].siz; ++i) if(node[node[cur].ch[i]].siz > 0)
        dfs(node[cur].ch[i]);
}
```

在每一次匹配成功时，需要输出产生式、记录当前节点的标志 **tag**、记录语法树当前节点的儿子信息，以第一条产生式为例：

```
CompUnits: CompUnit{
    fprintf(f2, "CompUnits -> CompUnit\n");
    node[$$ = ++tot].siz = 0;
    node[$$].ch[node[$$].siz++] = $1;
    strcpy(tag[$$], "CompUnits");
}
| CompUnits CompUnit{
    fprintf(f2, "CompUnits -> CompUnits CompUnit\n");
    node[$$ = ++tot].siz = 0;
    node[$$].ch[node[$$].siz++] = $1;
    node[$$].ch[node[$$].siz++] = $2;
```

```
        strcpy(tag[$$], "CompUnits");  
    }  
    ;
```

而对于终结符，则在词法分析中更新其节点标签 `tag` 和 `siz = 0`

4. 语法错误处理

根据 2.sy 中的代码，我对以下几类语法分析中可能遇到的错误加入了错误处理：

常量定义错误 Constant definition error

 ConstDecl: CONST error ';'

变量定义错误 Variable definition error

 VarDecl: INT error ';'

函数参数错误 Function parameter error

 FuncDef: INT error Block | VOID error Block

块错误 Block error

 Block: '{' error '}'

表达式错误 Statement error

 Stmt: error ';'

条件错误 Condition error

 Cond: error

三、语义分析

1. 基本结构和函数

定义了如下的数据结构和函数便于进行语义分析和目标代码的生成：
类型和变量(属性)以及符号表：

```
enum Type {Int, Constint, Arr, Fint, Fvoid};
struct Var{
    Type type;
    int val, offset;
    vector<int> dim;
};
vector<map<string, Var> > symbol;
```

属性节点：

```
struct State{
    int val, offset, offset_of_arr, quad;
    bool is_const, is_arr;
    string name;
    //针对函数参数以及 true/false list
    vector<bool> para_is_const, para_is_arr;
    vector<int> truelist, falselist, dim, para_val, para_offset, para_offset_of_arr;
    vector<string> para_name;
}sta[114514]
```

合并和回填操作：

```
vector<int> merge(const vector<int> &a, const vector<int> &b){
    vector<int> res;
    for(auto x : a) res.push_back(x);
    for(auto x : b) res.push_back(x);
    return res;
}

void backpatch(vector<int> &a, int pos){
    for(auto x : a) Assemble[x] += ".L" + to_string(pos) + "\n";
}
```

将变量移入寄存器, 和将寄存器移入内存, 注意分类讨论(是否是全局、常量、数组), 常量直接 mov 数值即可, 全局(由于过程中 offset 为负, 所以全局的 offset 记为 1)的变量或数组直接用 rip 寄存器和变量名定位, 局部的则用栈内偏移量定位。注意这里不能使用 eax 寄存器用于计算数组偏移量, 否则在数组元素的除、取模中 eax 寄存器存在冲突:

```
void var2reg(int x, const char* reg){
    char tmp[114];
    if(sta[x].is_const){
        sprintf(tmp, "\tmovl\t%d, %%s\n", sta[x].val, reg); Assemble.push_back(tmp);
    }
```

```

    }
    else if(sta[x].offset == 1){
        if(sta[x].is_arr){
            sprintf(tmp, "\tmovl\t%d(%rbp), %%ebx\n", sta[x].offset_of_arr);
Assemble.push_back(tmp);
            Assemble.push_back("\tcltq\n");
            Assemble.push_back("\tleaq\t0(, %rbx, 4), %rdx\n");
            sprintf(tmp, "\tleaq\t%s(%rip), %rbx\n", sta[x].name.c_str());
Assemble.push_back(tmp);
            sprintf(tmp, "\tmovl\t(%rdx, %rbx), %%s\n", reg); Assemble.push_back(tmp);
        }
        else{
            sprintf(tmp, "\tmovl\t%s(%rip), %%s\n", sta[x].name.c_str(), reg);
Assemble.push_back(tmp);
        }
    }
    else{
        if(sta[x].is_arr){
            sprintf(tmp, "\tmovl\t%d(%rbp), %%ebx\n", sta[x].offset_of_arr);
Assemble.push_back(tmp);
            Assemble.push_back("\tcltq\n");
            sprintf(tmp, "\tmovl\t%d(%rbp, %rbx, 4), %%s\n", sta[x].offset, reg);
Assemble.push_back(tmp);

        }
        else{
            sprintf(tmp, "\tmovl\t%d(%rbp), %%s\n", sta[x].offset, reg);
Assemble.push_back(tmp);
        }
    }
}

void reg2var(const char* reg, int x){
    char tmp[114];
    if(sta[x].is_const){
        yyerror("Const Error");
        exit(0);
    }
    if(sta[x].offset == 1){
        if(sta[x].is_arr){
            sprintf(tmp, "\tmovl\t%d(%rbp), %%ebx\n", sta[x].offset_of_arr);
Assemble.push_back(tmp);
            Assemble.push_back("\tcltq\n");
            Assemble.push_back("\tleaq\t0(, %rbx, 4), %rdx\n");
            sprintf(tmp, "\tleaq\t%s(%rip), %rbx\n", sta[x].name.c_str());
Assemble.push_back(tmp);

```



```

        sprintf(tmp, "\tmovl\t%%s, (%rdx, %rbx)\n", reg); Assemble.push_back(tmp);
    }
    else{
        sprintf(tmp, "\tmovl\t%%s, %s(%rip)\n", reg, sta[x].name.c_str());
        Assemble.push_back(tmp);
    }
}
else{
    if(sta[x].is_arr){
        sprintf(tmp, "\tmovl\t%d(%rbp), %%ebx\n", sta[x].offset_of_arr);
        Assemble.push_back(tmp);
        Assemble.push_back("\tcltq\n");
        sprintf(tmp, "\tmovl\t%%s, %d(%rbp, %rbx, 4)\n", reg, sta[x].offset);
        Assemble.push_back(tmp);
    }
    else{
        sprintf(tmp, "\tmovl\t%%s, %d(%rbp)\n", reg, sta[x].offset);
        Assemble.push_back(tmp);
    }
}
}
}

```

将参数 mov/lea 进寄存器，后者是用于 scanf 的操作。类似上面需要分类讨论：

```

void para2reg(int x, int pos, const char *reg);
void lea_para2reg(int x, int pos, const char *reg);

```

调用函数和从函数中返回时的固定语句：

```

void call_func_push(){
    Assemble.push_back("\tpushq\t%rbp\n");
    Assemble.push_back("\tpushq\t%r8\n");
    Assemble.push_back("\tpushq\t%r9\n");
    Assemble.push_back("\tmovq\t%rsp, %rbp\n");
}

void ret_func_pop(){
    Assemble.push_back("\tpopq\t%r9\n");
    Assemble.push_back("\tpopq\t%r8\n");
    Assemble.push_back("\tpopq\t%rbp\n");
    Assemble.push_back("\tret\n");
}

```

2. 常量定义

支持常量型 int 变量的定义(但不支持数组的定义，我尝试新设置了一个 const array 类型然后用和下面变量数组定义相同的方式进行定义，但跑出来的结果是只能正确存储

但无法正确读取)

同时为了便于处理，将数组部分的定义改用了右递归。常量的定义需满足赋的值必须是常数。注意按全局和局部分类讨论还有重名问题。

```
ConstArr: /*empty*/ {$$ = 0;}
| '[' ConstExp '[' ConstArr{
    if($4) sta[$$ = $4].val = sta[$2].val * sta[$4].val;
    else sta[$$ = ++tot].val = sta[$2].val;
    sta[$$].dim.push_back(sta[$2].val);
}
;

ConstDef: ID ConstArr '=' ConstInitVal{
    if(symbol[level].find(tag[$1]) != symbol[level].end()){
        yyerror("Const Redefinition Error");
        exit(0);
    }
    char tmp[114];
    if(!level){
        if(!sta[$4].is_const){
            printf("Const Definition Error\n");
            exit(0);
        }
        if (!$2){
            Assemble.push_back("\t.section\t.rodata\n");
            Assemble.push_back("\t.align\t4\n");
            sprintf (tmp, "\t.type\t%s, @object\n", tag[$1]);
            Assemble.push_back(tmp);
            sprintf (tmp, "\t.size\t%s, 4\n", tag[$1]); Assemble.push_back(tmp);
            sprintf (tmp, "%s:\n", tag[$1]); Assemble.push_back(tmp);
            sprintf (tmp, "\t.long\t%d\n", sta[$4].val); Assemble.push_back(tmp);
            Assemble.push_back("\t.text\n");
            symbol[0][tag[$1]] = Var(Constint, sta[$4].val, 1);
        }
        //else ?
    }
    else{
        if (!$2){
            offset -= 4;
            Assemble.push_back("\tsubq\t$4, %rsp\n");
            sprintf (tmp, "\tmovl\t%d, %%edi\n", sta[$4].val);
            Assemble.push_back(tmp);
            sprintf (tmp, "\tmovl\t%%edi, %d(%rbp)\n", offset);
            Assemble.push_back(tmp);
            symbol[level][tag[$1]] = Var(Constint, sta[$4].val, offset);
        }
        //else ?
    }
}
```

```
}  
  
}  
;
```

3. 变量定义

先判断是否重名，然后分类讨论：

(1) 未赋值

理论上似乎应该分配到.bss 段，但为了方便还是统一放在.data 段

全局变量/数组：计算相应的 size，对于变量初值设为.long 0 或.zero 4 即可，而对于数组初始化值为.zero 4 * size

以全局变量为例：

```
sprintf (tmp, "\t.globl\t%s\n", tag[$1]); Assemble.push_back(tmp);  
Assemble.push_back("\t.data\n");  
Assemble.push_back("\t.align\t4\n");  
sprintf (tmp, "\t.type\t%s, @object\n", tag[$1]); Assemble.push_back(tmp);  
sprintf (tmp, "\t.size\t%s, 4\n", tag[$1]); Assemble.push_back(tmp);  
sprintf (tmp, "%s:\n", tag[$1]); Assemble.push_back(tmp);  
Assemble.push_back("\t.long\t0\n");  
Assemble.push_back("\t.text\n");  
symbol[0][tag[$1]] = Var(Int, 0, 1);
```

局部变量/数组：栈上分配相应空间即可，还是以变量为例

```
offset -= 4;  
Assemble.push_back("\tsubq\t$4, %rsp\n");  
symbol[level][tag[$1]] = Var(Int, 0, offset);
```

(2) 赋值

首先我先针对数组的赋值定义了一个全局的 vector(ArrInitVal)来记录初始化的数组值，且用不同的返回值判断是变量还是数组的赋值(规定对于全局变量/数组的初始化赋值必须是常量)：

```
InitVals: InitVal {  
    if(!level && !sta[$1].is_const){  
        yyerror("Initializer Error");  
        exit(0);  
    }  
    ArrInitVal.push_back($1);  
}  
| InitVals ',' InitVal {  
    if(!level && !sta[$3].is_const){  
        yyerror("Initializer Error");  
        exit(0);  
    }  
    ArrInitVal.push_back($3);  
}
```

```

;

InitVal: Exp{ $$ = $1; }
    | '{' '}' { $$ = 0; }
    | '{' InitVals '}' { $$ = 0; }
;

```

全局变量/数组：根据 InitVal 的返回值情况判断类型是否匹配，另外对于数组还应判断赋值的个数是否超过了数组大小，其余部分类似上面，以数组为例：

```

if($4){
    yyerror("Initializer Error");
    exit(0);
}
if(ArrInitVal.size() > sta[$2].val){
    yyerror("Too Many Initializers Error");
    exit(0);
}
sprintf (tmp, "\t.globl\t%s\n", tag[$1]); Assemble.push_back(tmp);
Assemble.push_back("\t.data\n");
Assemble.push_back("\t.align\t32\n");
sprintf (tmp, "\t.type\t%s, @object\n", tag[$1]); Assemble.push_back(tmp);
sprintf (tmp, "\t.size\t%s, %d\n", tag[$1], sta[$2].val * 4); Assemble.push_back(tmp);
sprintf (tmp, "%s:\n", tag[$1]); Assemble.push_back(tmp);
for(auto x : ArrInitVal) {
    sprintf (tmp, "\t.long\t%d\n", sta[x].val); Assemble.push_back(tmp);
}
sprintf (tmp, "\t.zero\t%d\n", sta[$2].val * 4 - ArrInitVal.size() * 4);
Assemble.push_back(tmp);
Assemble.push_back("\t.text\n");
symbol[0][tag[$1]] = Var(Arr, 0, 1, sta[$2].dim);
ArrInitVal.clear();

```

局部变量/数组：同样类似非赋值的部分，注意这里的赋值可能不是常数，需要按照地址通过 var2reg 函数计算，然后放置到相应的地址位置，以数组为例：

```

if($4){
    yyerror("Initializer Error");
    exit(0);
}
if(ArrInitVal.size() > sta[$2].val){
    yyerror("Too Many Initializers Error");
    exit(0);
}
offset -= 4 * sta[$2].val;
sprintf (tmp, "\tsubq\t%d, %rsp\n", 4 * sta[$2].val); Assemble.push_back(tmp);
for(int i = 0; i < ArrInitVal.size(); ++i){
    var2reg(ArrInitVal[i], "edi");
}

```

```

        sprintf (tmp, "\tmovl\t%%edi, %d(%rbp)\n", offset + i * 4); Assemble.push_back(tmp);
    }
    symbol[level][tag[$1]] = Var(Arr, 0, offset, sta[$2].dim);
    ArrInitVal.clear();

```

4. 函数定义

以返回值为 int 类型的含参函数为例：

对于参数部分，在匹配到函数名后，匹配参数前需将参数列表清空，每次匹配到一个参数时将其 **push_back** 到参数列表中

```

INT FName '(' FuncFParams ')' Entry_Int_Para Block
FName: ID {
    strcpy(funcname, tag[$1]);
    Para.clear();
}
;
FuncFParams: FuncFParam {}
| FuncFParams ',' FuncFParam{}
;
FuncFParam: INT ID ParaArr{
    Para.push_back(tag[$2]);
}
;

```

进入函数前，新建一个符号表并将参数定义在符号表中，同时将寄存器和参数压入栈：

```

Entry_Int_Para: /*empty*/ {
    is_func_void = false;
    for(int i = level; i >= 0; --i)
        if(symbol[i].find(funcname) != symbol[i].end()){
            yyerror("Function Redefinition Error");
            exit(0);
        }
    symbol[level][funcname] = Var(Fint, 0, 0);
    ++level;
    map<string, Var> x;
    symbol.push_back(x);
    char tmp[114];
    sprintf(tmp, "\t.globl\t%s\n", funcname); Assemble.push_back(tmp);
    Assemble.push_back("\t.type\tmain, @function\n");
    sprintf(tmp, "%s:\n", funcname); Assemble.push_back(tmp);
    call_func_push();
    for(int i = 0; i < Para.size(); ++i) {
        offset -= 4;
        Assemble.push_back("\tsubq\t$4, %rsp\n");
    }
}

```

```

        sprintf(tmp, "\tmovl\t%d(%rbp), %r8d\n", 32 + i * 4);
Assemble.push_back(tmp);
        sprintf(tmp, "\tmovl\t%r8d, %d(%rbp)\n", offset); Assemble.push_back(tmp);
        symbol[level][Para[i].c_str()] = Var(Int, 0, offset);
    }
}
;

```

函数结束后，将栈恢复，弹出旧寄存器值

```

--level;
symbol.pop_back();
char tmp[114];
sprintf(tmp, "\taddq\t%d, %rsp\n", -offset); Assemble.push_back(tmp);
ret_func_pop();
offset = 0, is_func_void = false;

```

5. 条件控制

修改语法如下：

```

IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt %prec WITHOUTELSE
IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt ELSE AfterElse NewLabel BeforeStmt
Stmt AfterStmt NewLabel
WHILE WhileBegin BeforeStmt '(' Cond ')' WhileEnd NewLabel Stmt AfterStmt

```

其中 Newlabel 函数用于新建一个标签，BeforeStmt 是为进入新一层的嵌套作准备，即新建符号表并对齐，而 AfterStmt 则是从上一层的嵌套中恢复偏移量。

```

NewLabel: /*empty*/ {
    $$ = ++num;
    char tmp[114];
    sprintf (tmp, ".L%d:\n", num); Assemble.push_back(tmp);
}
;
BeforeStmt: /*empty*/ {
    ++level;
    map<string, Var> x;
    symbol.push_back(x);
    p_offset.push_back(offset);
    if(abs(offset) % 16 != 0){
        int padding = 16 - abs(offset) % 16;
        offset -= padding;
        char tmp[114];
        sprintf (tmp, "\tsubq\t%d, %rsp\n", padding); Assemble.push_back(tmp);
    }
}
;

```

```

AfterStmt: /*empty*/ {
    char tmp[114];
    int last = *p_offset.rbegin();
    sprintf (tmp, "\taddq\t%d, %rsp\n", last - offset); Assemble.push_back(tmp);
    p_offset.pop_back();
    offset = last;
}
;

```

对于 if 和 if-else 语句，可以通过使用 NewLabel 可以更加直观地完成回填操作(其中不含 else 的 if 语句尾部的新标签单独写在了最后的语义动作中)。AfterElse 则用于执行完 true 的部分后新增一条 jmp 指令：

```

IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt %prec WITHOUTELSE {
    --level;
    symbol.pop_back();
    char tmp[114];
    ++num;
    sprintf (tmp, ".L%d:\n", num); Assemble.push_back(tmp);
    for(auto x : sta[$3].truelist) Assemble[x] += ".L" + to_string($5) + "\n";
    for(auto x : sta[$3].falselist) Assemble[x] += ".L" + to_string(num) + "\n";
}
IF '(' Cond ')' NewLabel BeforeStmt Stmt AfterStmt ELSE AfterElse NewLabel BeforeStmt
Stmt AfterStmt NewLabel{
    --level;
    symbol.pop_back();
    for(auto x : sta[$3].truelist) Assemble[x] += ".L" + to_string($5) + "\n";
    for(auto x : sta[$3].falselist) Assemble[x] += ".L" + to_string($11) + "\n";
    Assemble[sta[$10].truelist[0]] += ".L" + to_string($15) + "\n";
}
AfterElse: /*empty*/ {
    Assemble.push_back("\tjmp\t");
    sta[$$ = ++tot].truelist.push_back(Assemble.size() - 1);
}
;

```

对于 while 循环，WhileBegin 标记每趟循环判断语句之前的位置，新建一个标签和 break/continue list 层，WhileEnd 则标记循环判断语句后的位置并恢复栈偏移量。

```

WhileBegin: /*empty*/ {
    $$ = ++num;
    char tmp[114];
    sprintf (tmp, ".L%d:\n", num); Assemble.push_back(tmp);
    vector< pair<int, int> > x, y;
    breaklist.push_back(x);
    continuelist.push_back(y);
}
;

```

```

WhileEnd: /*empty*/ {
    char tmp[114];
    sta[$$ = ++tot].quad = ++num;
    sprintf (tmp, ".L%d:\n", num); Assemble.push_back(tmp);
    int last = *p_offset.rbegin();
    sprintf (tmp, "\taddq\t%d, %rsp\n", last - offset); Assemble.push_back(tmp);
    Assemble.push_back("\tjmp\t");
    sta[$$].truelist.push_back(Assemble.size() - 1);
}
;
    sta[$$ = ++tot].truelist.push_back(Assemble.size() - 1);
}
;

```

对于 break 和 continue, 恢复栈内容后跳转:

```

WHILE WhileBegin BeforeStmt '(' Cond ')' WhileEnd NewLabel Stmt AfterStmt{
    --level;
    symbol.pop_back();
    backpatch(sta[$5].truelist, $8);
    char tmp[114];
    sprintf (tmp, "\tjmp\t.L%d\n", $2); Assemble.push_back(tmp);
    ++num;
    sprintf (tmp, ".L%d:\n", num); Assemble.push_back(tmp);
    backpatch(sta[$5].falselist, sta[$7].quad);
    backpatch(sta[$7].truelist, num);
    for(auto it : *breaklist.rbegin()){
        sprintf(tmp, "\taddq\t%d, %rsp\n", offset - it.second);
        Assemble[it.first - 1] = string(tmp);
        Assemble[it.first] += ".L" + to_string(num) + "\n";
    }
    breaklist.pop_back();
    for(auto it : *continuelist.rbegin()){
        sprintf(tmp, "\taddq\t%d, %rsp\n", offset - it.second);
        Assemble[it.first - 1] = string(tmp);
        Assemble[it.first] += ".L" + to_string($2) + "\n";
    }
    continuelist.pop_back();
}

```

6. 布尔表达式

比较操作, 将两个操作数放入寄存器中进行相应比较即可, 然后将分支加入 true/false list, 以<为例:

```

char tmp[114];
sta[$$ = ++tot].quad = ++num;

```



```

sprintf(tmp, ".L%d:\n", num); Assemble.push_back(tmp);
var2reg($1, "r8d");
var2reg($3, "r9d");
Assemble.push_back("\tcmp\tr8d, %r9d\n");
Assemble.push_back("\tj1\t");
sta[$$].truelist.push_back(Assemble.size() - 1);
Assemble.push_back("\tjge\t");
sta[$$].falselist.push_back(Assemble.size() - 1);

```

与、或操作则需要回填，以与为例：

```

backpatch(sta[$1].truelist, sta[$3].quad);
sta[$$].falselist = merge(sta[$1].falselist, sta[$3].falselist);
sta[$$].truelist = sta[$3].truelist;
sta[$$].quad = sta[$1].quad;

```

7. 左值表达式

讨论是否为数组进行判断，int 型变量/常量元素直接从符号表中读取即可，否则要计算数组偏移地址：

```

bool used = false;
if(!$2){
    for(int i = level; i >= 0; --i)
        if(symbol[i].find(tag[$1]) != symbol[i].end()){
            used = true;
            Var cur = symbol[i][tag[$1]];
            $$ = ++tot;
            if(cur.type == Constint) {
                sta[$$].is_const = true;
                sta[$$].val = cur.val;
            }
            else {
                sta[$$].is_const = false;
                sta[$$].offset = cur.offset;
                if(!i) sta[$$].name = string(tag[$1]);
            }
            break;
        }
}
else{
    char tmp[114];
    for(int i = level; i >= 0; --i)
        if(symbol[i].find(tag[$1]) != symbol[i].end()){
            Var cur = symbol[i][tag[$1]];
            if(cur.type != Arr || sta[$2].dim.size() != cur.dim.size()) continue;
            used = true;

```

```

int x = 1;
offset -= 4;
Assemble.push_back("\tsubq\t$4, %rsp\n");
sprintf (tmp, "\tmovl\t$0, %d(%rbp)\n", offset); Assemble.push_back(tmp);
int z = offset;
for(int j = 0; j < cur.dim.size(); ++j){
    var2reg(sta[$2].dim[j], "r8d");
    sprintf (tmp, "\timull\t%d, %r8d\n", x); Assemble.push_back(tmp);
    sprintf (tmp, "\taddl\t%d(%rbp), %r8d\n", z); Assemble.push_back(tmp);
    sprintf (tmp, "\tmovl\t%%r8d, %d(%rbp)\n", z); Assemble.push_back(tmp);
    x *= cur.dim[j];
}
$$ = ++tot;
sta[$$].is_arr = true;
if(cur.type == Constint){
    sta[$$].is_const = true;
    sta[$$].val = cur.val;
}
else{
    sta[$$].is_const = false;
    sta[$$].offset = cur.offset;
    sta[$$].offset_of_arr = z;
    if(!i) sta[$$].name = string(tag[$1]);
}
break;
}
}
if(!used){
    yyerror("Reference Undefined Variable");
    exit(0);
}

```

8. 函数调用

以有参数的函数调用为例，首先记录参数状态：

```

FuncRParams: Exp{
    $$ = $1;
    sta[$$].para_name.push_back(sta[$1].name);
    sta[$$].para_val.push_back(sta[$1].val);
    sta[$$].para_offset.push_back(sta[$1].offset);
    sta[$$].para_offset_of_arr.push_back(sta[$1].offset_of_arr);
    sta[$$].para_is_const.push_back(sta[$1].is_const);
    sta[$$].para_is_arr.push_back(sta[$1].is_arr);
}
| FuncRParams ',' Exp{

```

```

    $$ = $1;
    sta[$$].para_name.push_back(sta[$3].name);
    sta[$$].para_val.push_back(sta[$3].val);
    sta[$$].para_offset.push_back(sta[$3].offset);
    sta[$$].para_offset_of_arr.push_back(sta[$3].offset_of_arr);
    sta[$$].para_is_const.push_back(sta[$3].is_const);
    sta[$$].para_is_arr.push_back(sta[$3].is_arr);
}
;

```

随后对齐，将每个参数逐一插入栈中，有返回值的函数还应在调用后将 `eax` 寄存器的值存入栈中：

```

if(symbol[0].find(tag[$1]) == symbol[0].end()){
    yyerror("Funciton Undefined Error");
    exit(0);
}
char tmp[114];
if((- (offset - 4 * sta[$3].para_name.size())) % 16){
    int padding = 16 - abs(offset) % 16;
    offset -= padding;
    sprintf (tmp, "\tsubq\t%d, %rsp\n", padding); Assemble.push_back(tmp);
}
for(int i = sta[$3].para_name.size() - 1; i >= 0 ; --i){
    para2reg($3, i, "r8d");
    offset -= 4;
    Assemble.push_back("\tsubq\t4, %rsp\n");
    sprintf (tmp, "\tmovl\t%r8d, %d(%rbp)\n", offset); Assemble.push_back(tmp);
}
sprintf (tmp, "\tcall\t%s\n", tag[$1]); Assemble.push_back(tmp);
Var function = symbol[0][tag[$1]];
if(function.type == Fint){
    offset -= 4;
    Assemble.push_back("\tsubq\t4, %rsp\n");
    sprintf (tmp, "\tmovl\t%eax, %d(%rbp)\n", offset); Assemble.push_back(tmp);
    sta[$$ = ++tot].offset = offset;
}

```

9. 常规计算语句

如果操作数是常数则直接计算即可，否则转移至寄存器中进行相应计算后再存入内存，以加法为例：

```

if(sta[$1].is_const && sta[$3].is_const)
    sta[$$ = ++tot].val = sta[$1].val + sta[$3].val, sta[$$].is_const = 1;
else{
    char tmp[114];
    var2reg($1, "r8d");

```

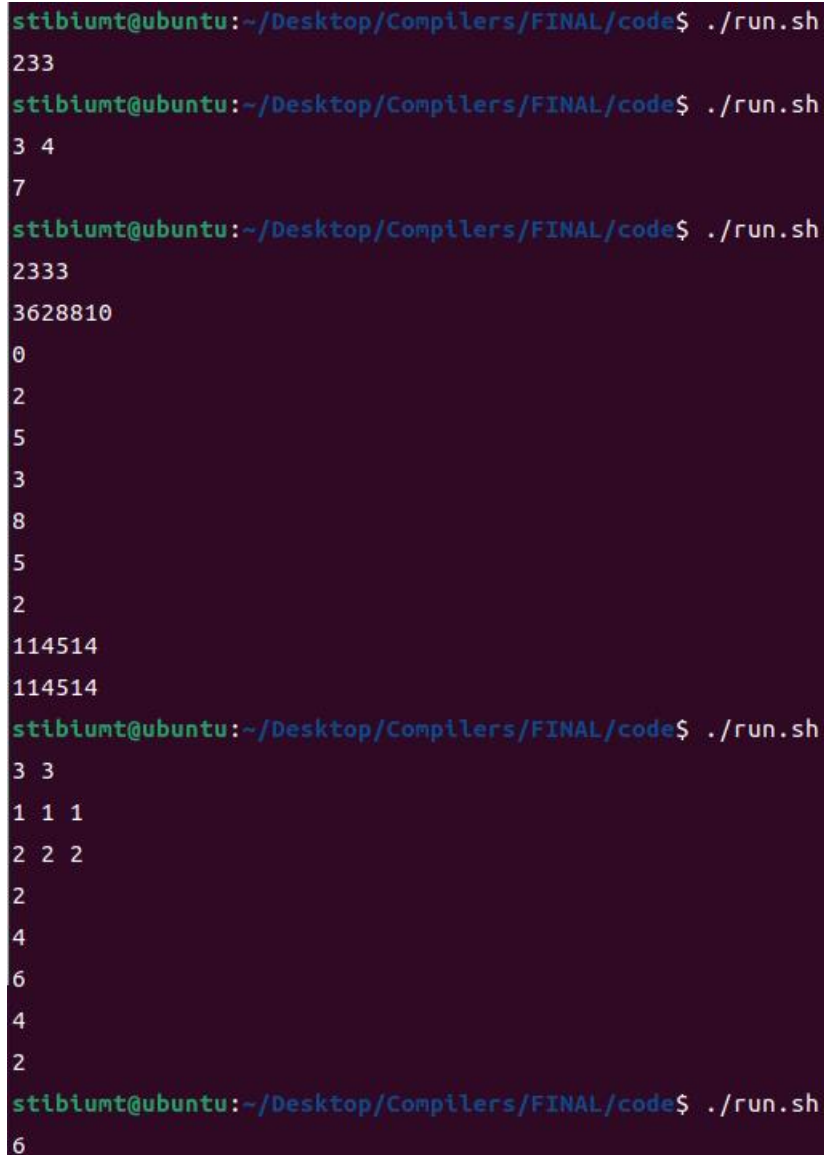
```
var2reg($3, "r9d");
Assemble.push_back("\taddl\t%r9d, %r8d\n");
offset -= 4;
Assemble.push_back("\tsubq\t$4, %rsp\n");
sprintf (tmp, "\tmovl\t%r8d, %d(%rbp)\n", offset); Assemble.push_back(tmp);
sta[$$ = ++tot].offset = offset;
}
```

四、 最终结果展示

以 test.sy 为例，运行脚本如下：

```
flex ./word.l
yacc -d ./grammar.y
g++ y.tab.c lex.yy.c -o mc -O2 -w
./mc ./test.sy
gcc assemble.s -o assemble
./assemble
```

test.sy(自行加入了一条输出语句)，1.sy-4.sy 的运行结果如下图：



```
stibiunt@ubuntu:~/Desktop/Compilers/FINAL/code$ ./run.sh
233
stibiunt@ubuntu:~/Desktop/Compilers/FINAL/code$ ./run.sh
3 4
7
stibiunt@ubuntu:~/Desktop/Compilers/FINAL/code$ ./run.sh
2333
3628810
0
2
5
3
8
5
2
114514
114514
stibiunt@ubuntu:~/Desktop/Compilers/FINAL/code$ ./run.sh
3 3
1 1 1
2 2 2
2
4
6
4
2
stibiunt@ubuntu:~/Desktop/Compilers/FINAL/code$ ./run.sh
6
```