实验报告

词法分析器的重写

首先我观察Sysy语言的语法规则，我发现其语法与关键字，运算符和界符的类型有关，因此我将词法分析的返回值进行修改，我将每一关键字和界符都设置为语法分析器中的一个终结符，并且对运算符通过优先级和结合性进行分类返回一个语法分析器中的一个终结符的类型。同时我将变量的标识符也作为终结符返回。以标识符为例，词法分析器返回的结构如下：

```
[a-zA-Z_][a-zA-Z0-9_]* {
    num_char += yyleng; yylval = ++cnt;
    strcpy(str[cnt], yytext);
    return Ident;
}
```

其中将识别出来的终结符的原字符串和其所在的位置，将其拷贝到语法分析器中的一个字符串数组中，同时将该字符串数组的基底指针加一。

语法分析器的实现

我观察Sysy语言的语法结构发现在该语法分析器中利用了正则表达式来表示语法的递归调用。例如对于常量数组变量的声明，我定义如下语法：

```
ConstDef:
    Ident array ASSIGN ConstInitVal {
        $$ = ++ cnt;
        strcpy(str[$$], "ConstDef");
            fprintf (fd, "%s %s %s %s -> ConstDef\n"
                , str[$1], str[$2], str[$3], str[$4]);
    }
    | error {
        $$ = ++ cnt;
        strcpy(str[$$], "ConstDef");
    }
;

array:
    array LBRACKET ConstExp RBRACKET {
        $$ = ++ cnt;
        strcpy(str[$$], "array");
            fprintf (fd, "%s %s %s %s -> array\n"
                , str[$1], str[$2], str[$3], str[$4]);
    }
    | empty {
        $$ = ++ cnt;
        strcpy(str[$$], "array");
            fprintf (fd, "empty -> array\n");
    }
;
```

我通过添加非终结符并且实现左递归来实现语法中的递归调用，同时对于运算符的优先级问题，我根据Sysy语法设置多个表达式的非终结符层层归约实现运算符的优先级问题。对于if-else归约的问题，我利用yacc中的%prec标识进行实现，代码如下：

```
| IF LPAREN Cond RPAREN Stmt %prec IF {
        $$ = ++ cnt;
        strcpy(str[$$], "Stmt");
            fprintf ("fd, %s %s %s %s %s -> Stmt\n"
                , str[$1], str[$2], str[$3], str[$4], str[$5]);
}
| IF LPAREN Cond RPAREN Stmt ELSE Stmt {
        $$ = ++ cnt;
        strcpy(str[$$], "Stmt");
            fprintf (fd, "%s %s %s %s %s %s %s -> Stmt\n"
}
```

同时对于每一个归约语法，我都将该非终结符写入字符串数组中并将归约步骤写入文件中，用于之后语法树的构建。

语法树的构建

在上述语法分析器对程序语法的分析中，我将每一次的归约项都写入了文件，因此从文件末尾到文件的开头就是语法树从上到下的一次dfs遍历，因此我构建如下Python脚本：

```
import re

def build_parsing_table(rules):
    l = []
    r = []
    for rule in rules:
        lhs, rhs = rule.split('->')
        l_list = lhs.split()
        rsh = rhs.replace(" ", "").replace("\n", "")
        l.append(l_list)
        r.append(rsh)
    return l, r

Non_terminator = ["CompUnit", "Decl", "ConstDecl", "ConstDefList", "ConstDef",
                  "array", "ConstInitVal", "ConstInitValList", "VarDecl",
"VarDefList",
                  "VarDef", "InitVal", "InitValList", "FuncDef", "FuncDef",
"FuncFParams",
                  "FuncFParam", "Block", "BlockItemList", "BlockItem", "Stmt",
                  "Exp", "Cond", "LVal", "PrimaryExp", "UnaryExp",
                  "FuncRParams", "MulExp", "AddExp", "RelExp", "EqExp",
                  "LAndExp", "LOrExp", "ConstExp"]

idx = 0
node = 1
def build_tree(fa, part, l, r):
```

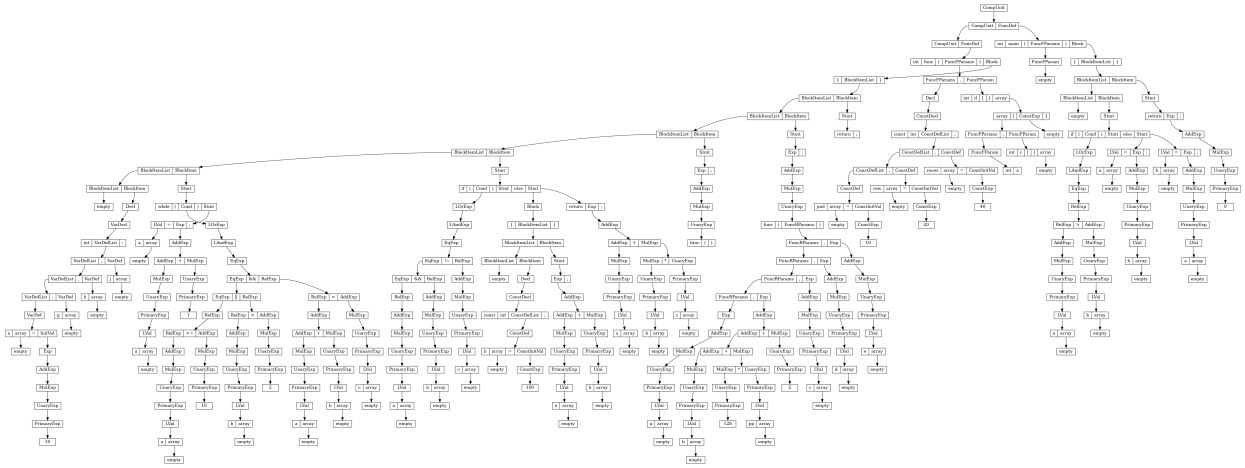
```
    global idx, Non_terminator, node
    if idx == len(l):
        return ;
    key = r[idx]
    value = l[idx]
    idx += 1
    cur = node
    node += 1
    info = "node{}[label = \"".format(cur)
    cnt = 0
    dict = {}
    for v in value:
        dict[v] = cnt
        if len(v) <= 2:
            v = ''.join([f"\\{c}" for c in v])
        info += "<f{}> ".format(cnt) + v + "|"
        cnt += 1;
    info = info[:-1]
    info += "\"];\n"
    g.write(info)

    g.write("\"node{}\":f{}->\"node{}\";\n".format(fa, part, cur))
    value.reverse()
    for v in value:
        if v in Non_terminator:
            build_tree(cur, dict[v], l, r)

with open('detail.txt') as f:
    rules = f.readlines()
    l, r = build_parsing_table(rules);
    l.reverse(), r.reverse();
    with open('graph.dot', 'w') as g:
        g.write("digraph \" \"{")
        g.write("node [shape = record,height=.1]")
        g.write("node0[label = \"<f0> CompUnit\"];")
        build_tree(0, 0, l, r)
        g.write("}")
```

该脚本可以倒序依次读取归约文件，并通过自顶向下的方式构建出语法树的一条边，并以Graphviz中需要的.dot文件的方式构建点对点的一条边。

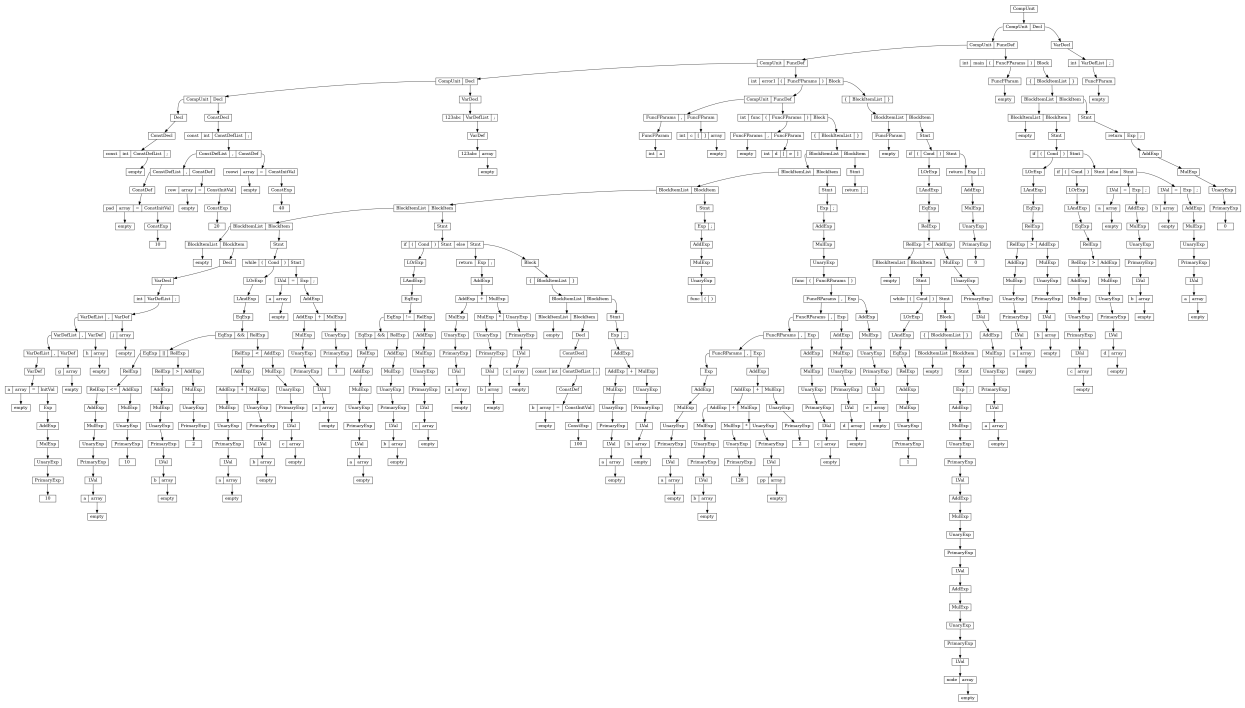
实验结果

通过上述实验，我构建的语法树如图所示：

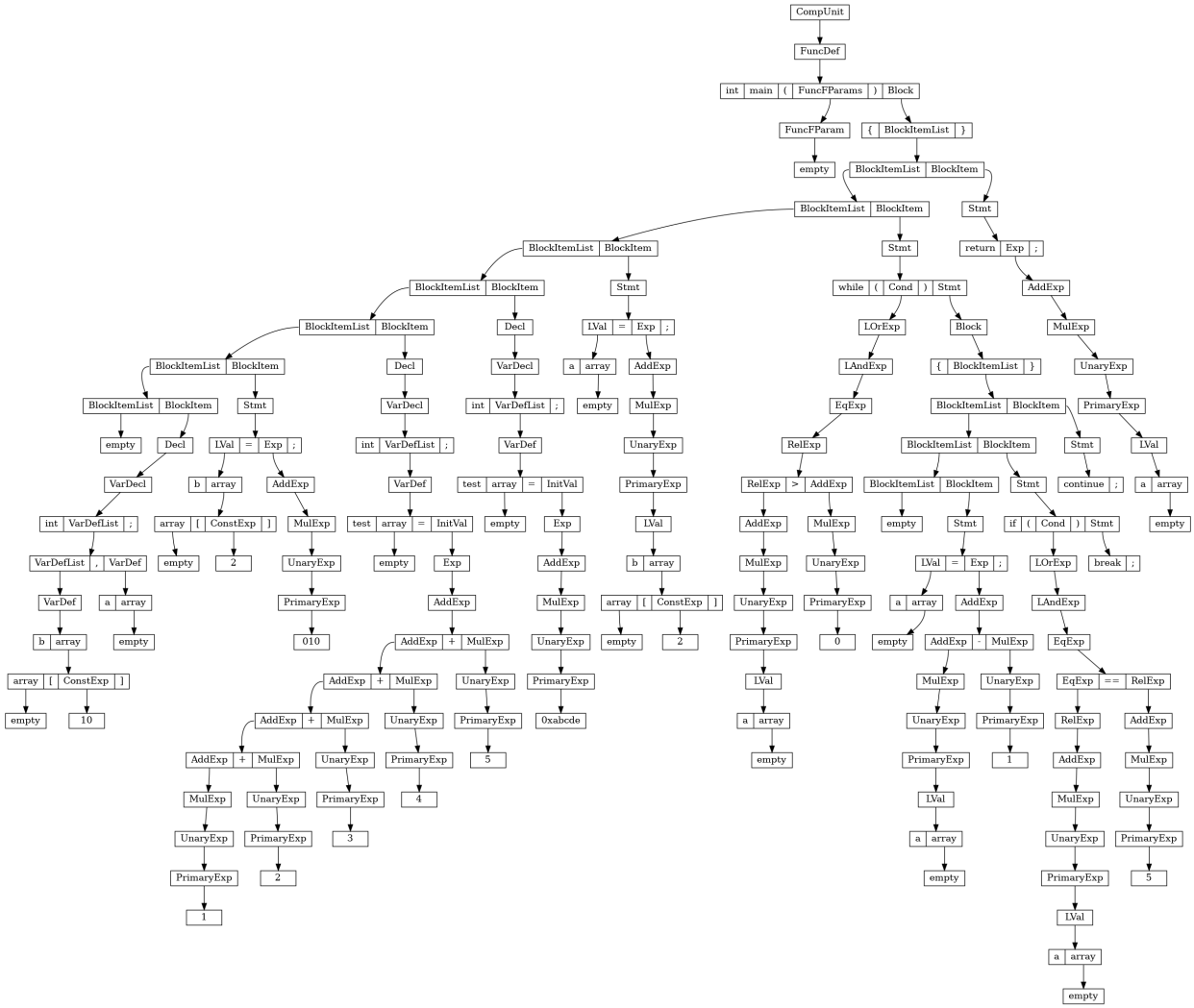
1.sy



2.sy



3.sy


CompUnit
FuncDef
int main (FuncFParams) Block
FuncFParam
{ BlockItemList }
empty
BlockItemList BlockItem
BlockItemList BlockItem
Stmt
BlockItemList BlockItem
Stmt
return Exp ;
BlockItemList BlockItem
BlockItemList BlockItem
Decl
LVal = Exp ;
while (Cond) Stmt
AddExp
BlockItemList BlockItem
Decl
VarDecl
a array
AddExp
LOrExp
Block
MulExp
BlockItemList BlockItem
Stmt
VarDecl
int VarDefList ;
empty
MulExp
LAndExp
{ BlockItemList }
UnaryExp
empty
Decl
LVal = Exp ;
int VarDefList ;
VarDef
empty
UnaryExp
EqExp
BlockItemList BlockItem
PrimaryExp
VarDecl
b array
AddExp
VarDef
test array = InitVal
PrimaryExp
RelExp
BlockItemList BlockItem
Stmt
LVal
int VarDefList ;
array [ConstExp]
MulExp
test array = InitVal
empty
Exp
LVal
RelExp > AddExp
BlockItemList BlockItem
Stmt
continue ;
a array
VarDefList , VarDef
empty
2
UnaryExp
empty
Exp
b array
AddExp
AddExp
MulExp
UnaryExp
empty
LVal = Exp ;
LOrExp
break ;
VarDef
a array
PrimaryExp
010
AddExp + MulExp
array [ConstExp]
MulExp
UnaryExp
empty
2
UnaryExp
PrimaryExp
0
a array
AddExp - MulExp
LAndExp
b array
empty
AddExp + MulExp
UnaryExp
PrimaryExp
0xabcde
empty
2
PrimaryExp
LVal
empty
MulExp
UnaryExp
EqExp
array [ConstExp]
10
AddExp + MulExp
UnaryExp
PrimaryExp
5
a array
empty
UnaryExp
PrimaryExp
1
EqExp == RelExp
AddExp + MulExp
UnaryExp
PrimaryExp
4
PrimaryExp
LVal
RelExp
AddExp
MulExp
UnaryExp
PrimaryExp
3
a array
empty
MulExp
UnaryExp
PrimaryExp
5
UnaryExp
PrimaryExp
2
PrimaryExp
LVal
1
a array
empty