实验报告

一、实验目标

在词法分析器的基础上，借助 yacc 工具实现一个语法分析器，要求编写适当的语义动作，能够按照规约顺序输出需要用到的规约规则，同时绘制 SysY 代码的语法树。

二、代码说明

1. 词法分析器的修改

为了匹配语法分析器，要对之前的词法分析程序做一些修改。每当匹配到终结符时，不再打印类别信息，而是返回语法分析器中对应的终结符符号，比如：

```
"int" {return INT;}        "+" {return PLUS;}
"main" {return MAIN;}      "-" {return MINUS;}
```

此外，为了生成语法树，需要为识别到的标识符和常数建立语法树结点，因此将 yylval 的类型由 int 转变为自定义的 TreeNode*类型，并且提前声明使用到的 TreeNode2 函数（在 yacc.y 中定义）。这里没有为其它终结符建立结点，因为它们的表示是固定的，所以选择在之后的语法分析程序中新建对应结点。

```
typedef struct TreeNode{
    int id;
    int type;
    int length;
    struct TreeNode* child;
    struct TreeNode* brother;
    char* name;
}TreeNode, *Tree;
#define YYSTYPE TreeNode*
#include "y.tab.h"
extern TreeNode* TreeNode2(int type);
```

```
{IDENTIFIER} {TreeNode* node = TreeNode2(1);
strcpy(node->name, yytext);
yylval = node;
return IDENT;}
{INTCONST}  {TreeNode* node = TreeNode2(1);
strcpy(node->name, yytext);
yylval = node;
return INTCONST;}
```

2. 语法规则的定义

语法规则的定义大部分按照文档直接编辑，但是对于部分正则式，需要进行修改，比如

常量声明　　ConstDecl → **'const'** BType ConstDef { ',' ConstDef } ';'

需要变为

```
ConstDecl: CONST INT ConstDefs SEMI{$$ = TreeNode1("ConstDecl", 0);
```

```
ConstDefs: ConstDef{$$ = TreeNode1("ConstDefs", 0); addChild($$, $1);}
|ConstDefs COMMA ConstDef{$$ = TreeNode1("ConstDefs", 0);
```

规约规则的输出，直接在语义动作中 printf 即可，会按照规约顺序输出。

3. 语法树的建立

（1）结点定义

之前已经提到，我将 yylval 设置为自定义的 TreeNode*类型，所以语法规则中每个匹配上的非终结符和标识符、常数都是 TreeNode*类型。

```
typedef struct TreeNode{
    int id;  //结点的序号，用于在语法树中输出
    int type; //为1表示是叶子节点，0表示非叶结点
    int length; //是节点的第几部分
    struct TreeNode* child; //子节点
    struct TreeNode* brother; //兄弟结点
    char* name;  //存放结点的值，即对应的（非）终结符
}TreeNode, *Tree;
```

其中，id 对应 Tree.dot 文件中的节点名称 NAME_NODE；length 对应节点每个部分的标号<f >；name 对应结点每部分的名称 NAME

- 定义节点：

  中括号外为结点的名称（不会显示在图片中）。

  中括号中的双引号内部为结点每个部分的名称。尖括号中为该部分的标号，后续的空格为该部分的名称（显示在图片中的）

```
1  NAME_NODE[label = "<f0> NAME0|<f1> NAME1"];
```

（2）语法树结构

语法树为每个匹配到的终结符和规约得到的非终结符都建立结点，每当匹配上语法规则后，就将规约元素作为根节点，匹配上的（非）终结符作为子节点（第一

个匹配上的作为子节点，之后匹配上的作为子节点的兄弟节点）。即横向的兄弟节点链表存放规约元素的规约序列。举例来说，语法规则 ConstDecl->CONST INT ConstDefs SEMI 中，ConstDecl 是根节点，const 是其子节点，int 是 const 的兄弟节点，ConstDefs 是 int 的兄弟节点，’;’是 ConstDefs 的兄弟节点。然后非终结符 ConstDefs 又可以在其他语法规则中作为根节点，以此类推，就得到语法树的结构：纵向的父子结点表示规约，横向的兄弟节点链表存储规约序列。

```
ConstDecl: CONST INT ConstDefs SEMI{
```

（3）与语法树相关的函数：

- TreeNode1 函数在给定 name 和 type 的情况下新建节点，对其进行初始化并将其返回

```
TreeNode* TreeNode1(char* name, int type){
    TreeNode* root = (TreeNode*)malloc(sizeof(TreeNode));
    root->id = 0;
    root->type = type;
    root->name = (char*)malloc(sizeof(char)*20);
    strcpy(root->name, name);
    root->child = NULL;
    root->brother = NULL;
    return root;
}
```

- TreeNode2 函数与 TreeNode1 相同，只不过不提供 name

```
TreeNode* TreeNode2(int type){
    TreeNode* root = (TreeNode*)malloc(sizeof(TreeNode));
    root->id = 0;
    root->type = type;
    root->child = NULL;
    root->brother = NULL;
    root->name = (char*)malloc(sizeof(char)*20);
    return root;
}
```

- addChild 函数为 root 结点添加子节点 child，如果该节点已经有子节点，则将 child 添加为子节点的兄弟节点

```
void addChild(TreeNode* root, TreeNode* child){
        if(root->child == NULL){
            root->child = child;
        }else{
            addBrother(root->child, child);
        }
    }
```

- addBrother 函数为 root 添加兄弟节点 brother（在链表最后）

```
void addBrother(TreeNode* root, TreeNode* brother){
        TreeNode* p = root;
        while(p->brother != NULL){
            p = p->brother;
        }
        p->brother = brother;
    }
```

- createID 函数在语法树完整建立后遍历所有节点，分配 id 值。因为语法树自底向上建立，所以需要在建立完成之后才分配序号。

```
void createID(TreeNode* root){
        root->id = nodeid++;
        if (root->child)
            createID(root->child);
        if (root->brother)
            createID(root->brother);
    }
```

- changeID 函数是为了语法树图片的美观设立，将每个规约序列的 id 值变成相同的（第一个结点的 id），即在语法树中表示为一个节点。将 length 设置为每个符号在规约序列中所处的位置

```
void changeID(TreeNode* root){
        TreeNode* p = root->brother;
        while(p){
            p->id = root->id;
            p = p->brother;
        }
        if (root->child)
            changeID(root->child);
        if (root->brother)
            changeID(root->brother);
    }
```

- Traverse 函数对语法树进行先序遍历，输出结点和结点之间的连边关系到文件 Tree.dot

```
void Traverse(TreeNode* root, FILE* fp){
    TreeNode* p = root->child;
    if(p){
    fprintf(fp, "node%d[label = \"", p->id);
    int num = 0;
    while(p){
        if(num) fprintf(fp,"|");
        p->length = num;
        fprintf(fp, "<f%d> %s", num++, p->name);
        p = p->brother;
    }
    p = root->child;
    fprintf(fp, "\"];\n");
    fprintf(fp, "\"node%d\":f%d->\"node%d\";\n", root->id, root->length, p->id);
    }
    if (root->child)
        Traverse(root->child, fp);
    if (root->brother)
        Traverse(root->brother, fp);
}
```

- Print 函数调用 Traverse 函数，按照 graphviz 的规则输出到 Tree.dot 文件

```
void Print(TreeNode* root){
    createID(root);
    changeID(root);
    FILE* fp = fopen("Tree.dot", "w");
    root->length = 0;
    if(fp != NULL){
        fprintf(fp, "digraph \" \"\{\n");
        fprintf(fp, "node [shape = record,height=.1]\n");
        fprintf(fp, "node%d[label = \"<f0> CompUnit\"];\n", root->id);

        Traverse(root, fp);
        fprintf(fp, "\}");
        fclose(fp);
    }
}
```

- Clean 函数释放 malloc 申请的空间

```
void Clean(TreeNode* root){
    if(root->child)
        Clean(root->child);
    if(root->brother)
        Clean(root->brother);
    free(root->name);
    free(root);

}
```

（4）语义动作中语法树的建立

用一个例子来说明在语法规则的语义动作中如何建立语法树：首先为规约元素 ConstDecl 创建非叶结点，并为三个终结符‘const’、‘int’和‘;’创建叶节

点；接着将规约序列“const int ConstDefs ;”对应的四个节点依次添加为 ConstDecl 的子节点，最后输出对应的规约规则即可。

```
ConstDecl: CONST INT ConstDefs SEMI{$$ = TreeNode1("ConstDecl", 0);
TreeNode* tmp1 = TreeNode1("const", 1);
TreeNode* tmp2 = TreeNode1("int", 1);
TreeNode* tmp3 = TreeNode1("\\;", 1);
addChild($$, tmp1);
addChild($$, tmp2);
addChild($$, $3);
addChild($$, tmp3);
printf("ConstDecl -> const int ConstDef {, ConstDef} ;\n");}
;
```

4.注意事项

（1）if-else 的处理

对 if-else 语句中的移进-规约冲突，做如下优先级处理即可：

```
%nonassoc IFX
%nonassoc ELSE
```

```
IF LP Cond RP Stmt %prec IFX{$
```

（2）四则运算优先级

做如下定义，使‘*’‘/’的优先级比‘+’‘-’更高

```
%left PLUS MINUS
%left TIMES DIVIDE
```

(3)错误恢复

重新定义 yyerror 函数，当遇到错误时输出对应的错误和所在行号。

```
void yyerror(char* s){
    printf("Error:%s at line %d", s, num_lines);
}
```

三、结果展示

程序可以正确运行，没有移进-规约冲突。应用于测试样例 test.sy，得到测试样例按顺序的规约(较长，只展示一小部分)和语法树。

```
lixinyu@LAPTOP-OGLG3R4D:~$ yacc -d yaccn.y
lixinyu@LAPTOP-OGLG3R4D:~$ gcc y.tab.c lex.yy.c -o mc -O2 -w
lixinyu@LAPTOP-OGLG3R4D:~$ ./mc < test.txt > test.out
lixinyu@LAPTOP-OGLG3R4D:~$ dot -Tpng -o Tree.png Tree.dot
```

```
Number -> IntConst
PrimaryExp -> Number
UnaryExp -> PrimaryExp
MulExp -> UnaryExp
AddExp -> MulExp
ConstExp -> AddExp
ConstInitVal -> ConstExp
ConstDef -> Ident = ConstInitVal
ConstDecl -> const int ConstDef {, ConstDef} ;
Decl -> ConstDecl
CompUnit -> Decl
LVal -> Ident {'['Exp']'}
PrimaryExp -> LVal
UnaryExp -> PrimaryExp
MulExp -> UnaryExp
AddExp -> MulExp
ConstExp -> AddExp
VarDef -> Ident {'['ConstExp']'}
VarDecl -> int VarDef {, VarDef} ;
Decl -> VarDecl
CompUnit -> CompUnit Decl
```