词法分析与语法分析

学号：221900398

姓名：甘思毅

数据结构——树

为了能够存储语法中的终结符与非终结符，同时保存可能的值，设计如下的树节点结构：

```
struct Node {
    char *name;     // 节点名称
    syn_state type; // 节点类型
    int lineno;     // 节点第一次出现时的行数
    int childno;    // 此节点拥有的子节点数目
    union {         // 此节点具有的值
        int ival;
        float fval;
        char str[32];
    } value ;
    Node* children[MAX_CHILD]; // 子节点数组
};
```

节点：节点可以有多种类型：顶层的非终结符，终结符（包括ID、INT、FLOAT 等含有取值的词素），甚至是空串。因此设计一个枚举类型来区分不同的节点的类型：

```
enum Syn_state {
    SYN_NORMAL = 1, // 非终结符
    SYN_NULL,       // 空串
    SYN_INT,        // 终结符，整型词素
    SYN_FLOAT,      // 终结符，浮点型
    SYN_ID,         // 名称，含有字符串的值
    SYN_TYPE,       // TYPE 类信号
    SYN_TOKEN       // 其他终结符
};
```

在构造一个新节点的时候，因为节点的孩子数量是不确定的，因此引入 <stdarg.h> 辅助完成节点的构造：

```
#include <stdarg.h>

Node* createNode(char *name, syn_state state, int lineno, int childno, ...){
    ...
}
```

空串：在自顶向下的语法分析中，面临空串的向上传递问题：一个非终结符向下推导，可能得到一个空串作为终结符。在语法分析完成之后，需要对语法树进行"剪枝"：删除空串的分支

一个分支被删除的条件是:

1. 它是一个空串
2. 它是一个非终结符，但递归的查询其所有子节点，不含非空串的子节点

函数 remove_NULL_child() 进行剪枝：

```
bool remove_NULL_child(Node* node){
    if(node->type == SYN_NULL){
        return true;
    } else if(node->childno == 0){
        return false;
    }
    bool flag = true;
    for(int i=0; i < node->childno; i++){
        flag = flag && remove_NULL_child(node->children[i]);
    }
    if(flag){
        node->type = SYN_NULL;
        return true;
    } else {
        return false;
    }
}
```

flex 词法分析

选做：八进制、十六进制整数

```
digit   [0-9]
dec     [1-9]{digit}*|0
hex     0[xX][0-9a-fA-F]+
oct     0[0-7]+
```

我选择在词法分析的步骤将整数、浮点数等的值赋给对应的树节点。在词法分析的正则表达式中，完成数值的类型转换与树节点构造

```
{oct}   { yylval.node = createNode("INT", SYN_INT, yylineno, 0);
        yylval.node->value.ival = ato_oct(yytext); return INT; }
{hex}   { /* 类似于八进制 */}
// 同时处理非法的整数输入
0[0-7]*[89]+{digit}*    {
    printf("Error type A at Line %d: illegal oct: \'%s\'\n", yylineno,
yytext);
    lex_error += 1;}
0[xX][0-9a-fA-F]*[g-zG-Z]+({digit}|{letter})*   {
    printf("Error type A at Line %d: illegal hex: \'%s\'\n", yylineno,
yytext);
    lex_error += 1;}
```

bison 语法分析

在我的设计中，语法分析中的所有语法单元都应视为树的节点。通过 bison 语法特性来做到这一点：

```
%union {
    struct Node* node;
}
%type <node> CompSt StmtList Stmt DefList Def DecList Dec Exp Args
```

这样在语法分析时，可以直接将匹配的语法单元视为 Node 型数据

为了捕捉语法中的错误，重写 yyerror() 函数并在语法分析中使用。下面用 FunDec 举例：

```
FunDec  : ID LP VarList RP      {$$=createNode("FunDec", SYN_NORMAL, @$.first_line, 4, $1, $2, $3, $4); }
        | ID LP RP      {$$=createNode("FunDec", SYN_NORMAL, @$.first_line, 3, $1, $2, $3); }
        | ID LP error RP        {char buffer[64]; sprintf(buffer, "FunDec error: %s", yytext);                                          yyerror(buffer); syn_error++; yyerrok; }
        ;
```

程序编译

使用提供的默认后台 Makefile 即可

```
make
```

手动编译指令：

```
bison -d syntax.y
flex lexical.l
gcc main.c syntax.tab.c tree.c -lfl -o parser
```