2022年春季学期
哈尔滨工业大学计算学部
《编译原理》课程

Lab1 词法分析与语法分析 实验报告

目录

姓名	杨文昊
学号	1190303027
班号	1903202
电子邮件	675451361@qq.com
手机号码	15855161066

1. 亮点1: 词法分析完成所有选做, 并精准报错

将`常见错误`用正则表达式表示, 有`针对性的报错`, 使得报错内容更为精准

1.1 八进制数/十六进制数:

- `声明` : 八进制/十六进制 常见错误

```
wroct       0{digit_8}*[89]+{digit_8}*
wrhex       0[Xx]{digit_16}*[g-zG-Z]+{digit_16}*
```

- `动作` : 八进制/十六进制 常见错误的输出

```
{wroct}     { printf("Error type A at line %d: Illegal octal number \'%s\'\n", yylineno, yytext); out = false;  }
{wrhex}     { printf("Error type A at line %d: Illegal hexadecimal number \'%s\'\n", yylineno, yytext); out = false;  }
```

1.2 指数形式的浮点数:

- `声明` : 普通形式 常见错误

```
exphd       {dec}|{flo}
wrflo       ({exphd}[Ee]{flo})|({exphd}[Ee][^ \n\t;]*)|([Ee]{flo})
```

- `动作` : 浮点普通形式 常见错误的输出

```
{wrflo}     { printf("Error type A at line %d: Illegal float number \'%s\'\n", yylineno, yytext); out = false; }
```

1.3 多行注释:

- `声明` : 多行注释 常见错误

```
leftNoStar  \/\*[^*]*
leftStar    \/\*[^*]*\*+([^*/][^*]*\*+)*([^*/][^*]*)
left        {leftNoStar}|{leftStar}
right       \*\/
```

包括: leftNoStar, leftStar, 共同构成left错误; 缺失构成right错误

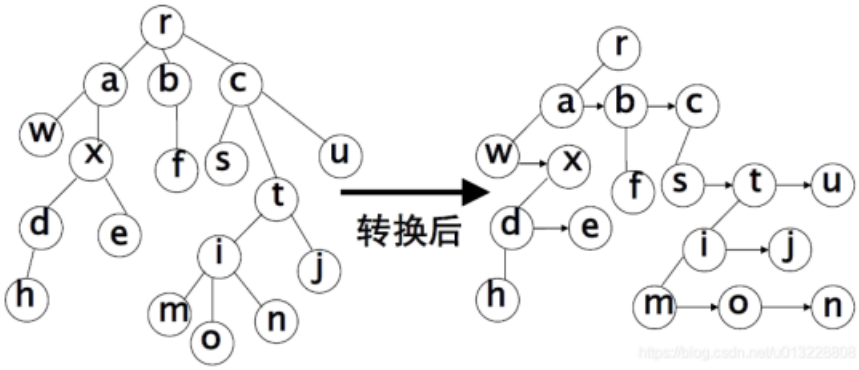
- `动作` : 多行注释 常见错误输出

```
{left}      { printf("Error type A at Line %d: unterminated annotaion \'%s\'", yylineno, yytext); out = false; }
{right}     { printf("Error type A at Line %d: alone end of multiline annotation \'%s\'\n", yylineno, yytext); out = false; }
```

2. 亮点2: 语法树构建与简洁的接口

2.1 语法树构建

语法树是一棵多叉树, 采用`二叉树`进行存储, 每个节点只有子节点和兄弟节点两个指针变量, 如下图:



封装出节点类型`ast`，同时在Bison源文件中将`yylval`定义成ast类型

```
struct ast {
    int line_no;                    // 行号
    enum Tag tag;                   // 节点的tag
    union {
        char str[STRING_LENGTH];
        int ival;
        float fval;
    } u;                            // 节点的值 (if exists)
    struct ast* first_child;        // 第一个子节点
    struct ast* first_sibling;      // 右兄弟
};
```

2.2 简洁的接口

在文件`ast.h`中, 定义的语法树的接口如下:

```
struct ast *newAst(enum Tag , int, ...);
void printAst(struct ast*, int);
char *outputTag(enum Tag tag);
```

采用C语言的`可变参数`传参, 并从...中依次取数据, 这样可以一次向父节点插入很多子节点, 极大程度降低了冗余性.

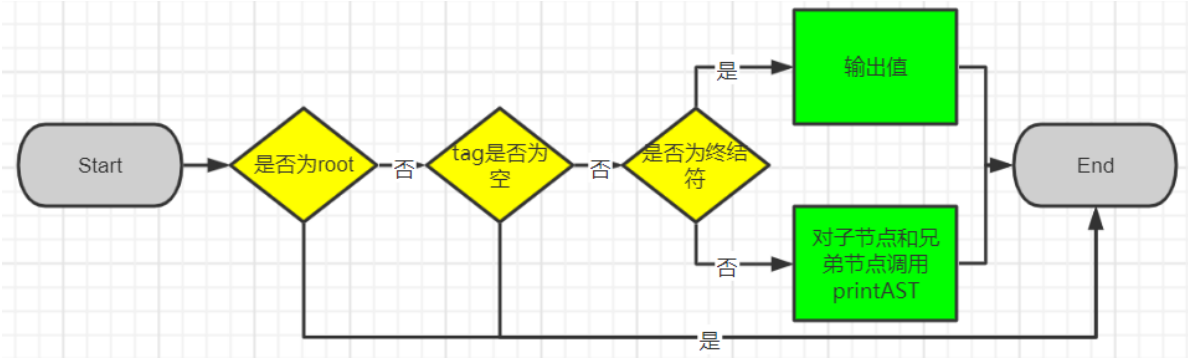
有了上述封装，我们在Bison源文件中定义词法单元。当我们在Flex源文件中进行词法分析时，创建ast节点并返回Bison源文件中相应的终结符词法单元。

而当我们在Bison源文件中进行语法分析时，我们在规则部分声明具体的C--语法并调用相应的创建插入动作即可。例如:

```
Exp:  Exp ASSIGNOP Exp   { $$ = newAst(TAG_EXP, 3, $1, $2, $3); }
```

2.3 递归实现newAST和printAST函数

newAST和printAST两者算法类似, 以printAST函数为例, 流程如下:



实现了简单但又清晰的调用, 在`main.c`中直接`printAst(astRoot, 0)`即可

```
if(out)
        printAst(astRoot, 0);
```

2.4 调用相应函数实现语法分析与报错

语法分析器与词法分析器协同:

- 对于每个终结符token，词法分析器生成对应类型的语法节点ast
- 同时进行语法分析匹配，对满足产生式右部的句子进行规约，规约成非终结符ast

生成终结符的ast节点: 以FLOAT为例

```
{FLOAT}      {
    struct ast *leaf = yylval.type_ast = newAst(TAG_FLOAT, 0,
yylineno);
    leaf->u.fval = atof(yytext);
    return FLOAT;
             }
```

产生式规约: 以函数声明为例:

```
FunDec: ID LP VarList RP    { $$ = newAst(TAG_FUN_DEC, 4, $1, $2,
$3, $4); }
    | ID LP RP              { $$ = newAst(TAG_FUN_DEC, 3, $1, $2,
$3); }
    ;
```

3. 编译与运行

- 编译: 在代码目录, 命令行输入`make`即可
- 运行: 编译完成后, 命令行输入`./parser xxx.cmm`即可