编译原理实验一：词法分析和语法分析

林搏海
哈尔滨工业大学 物联网工程

日期：2022 年 4 月 1 日

摘　　要

我们实现了基于 Flex 的词法分析和基于 Bison 的语法分析，构建了一颗相应的语法分析树并且将其可视化打印。在 Flex 语法分析中，我们实现了八进制和十六进制整型变量的识别，以及浮点数的科学计数法的识别。在构建语法分析树的过程中，我们使用了二叉树的结构来表示多叉树，并且在传参的过程中使用了 C 语言的变长参数函数。此外，我们采用了自底向上的过程来构建二叉树，正好对应于语法分析的 LR(1) 语法分析过程。

关键词： 语法分析树，二叉树，词法分析，Flex，Bison

1　词法分析

词法分析的实现过程采用了 Flex 模块。基于该模块，我们只需要构造识别所有类型单词单元的正则表达式即可，Flex 会将正则表达式自动生成 DFA 自动机程序。由于实验指导书中已经定义好了绝大部分词法单元的正则表达式，在此我们只需要生成 INT，FLOAT，和 ID 的正则表达式即可。需要注意的是，为了方便后续对于八进制和十六进制数的转换处理，我们将八进制数和十六进制树识别为不同于十进制数 INT 的类型 OCT 和 HEX。

在书写了正则表达式之后，我们需要将识别到的词法单元生成语法分析树的叶节点，方便后续完成自底向上的语法分析树的构建工作。需要注意的是，此实八进制数和十六进制数返回的叶节点类型都是 INT，但是其中叶节点的 value 值不同，这样就可以方便后续八进制值和十六进制值转换为十进制值的处理。

```
{ID}        { yylval.node = newNode(yylineno, TOKEN_ID, "ID", yytext, 0); return ID;}
{INT}       { yylval.node = newNode(yylineno, TOKEN_INT, "INT", yytext, 0); return INT;}
{OCT}       { yylval.node = newNode(yylineno, TOKEN_INT, "OCT", yytext, 0); return INT;}
{HEX}       { yylval.node = newNode(yylineno, TOKEN_INT, "HEX", yytext, 0); return INT;}
{FLOAT}     { yylval.node = newNode(yylineno, TOKEN_FLOAT, "FLOAT", yytext, 0); return FLOAT;}
```

图 1: 叶节点生成

最后需要注意的是，我们需要对输入文本中所有的空格进行处理，将其手动忽略，同时，我们需要在正则表达书写的最后部分加上通配符'.'，方便我们对错误的词法进行处理。

```
[\t\r\f ]+ {}
\n|\r       { yycolumn = 1;}
.           { lexError = TRUE; printf("Error type A at line %d: mysterious characters %s\n", yylineno, yytext);}
```

图 2: 空格和错误处理

2 语法分析

语法分析阶段的任务我们需要借助语法分析工具 Bison 来完成。该工具可以将我们书写的 LR 文法转换为对应的 PDA 自动机程序，由于 LR 文法表达式已经在实验指导书中被给出，因此，我们只需要在每一条文法的最后加入生成语法分析树节点的语义动作，并且插入 error 指令实现语法错误检测，最后再实现二义性文法的改造即可。

由于构造语法树的语句再词法分析和语法分析的阶段被大量使用，因此我们将其定义和是现在一个头文件中，然后我们只需要再对应的词法分析和语法分析文件中将其调用即可。在给出的 LR 文法中，我们发现有的父节点对应了多个子节点，因此可以考虑多叉树的实现，但是我们考虑更为简单的实现：将多叉树转换为二叉树。多叉树转换得到的二叉树的先序遍历和原本的多叉树完全相同这一性质，非常有利于我们在之后打印整个树的结构。此外，由于在构造父节点的时候，子节点的参数个数是不固定的，因此我们需要通过变长参数函数传递子节点，进行父节点的构造。

在打印树结构的时候，我们只需要按照二叉树的先序遍历递归打印即可，需要注意的是由于我们使用二叉树来标识多叉树结构，因此我们需要注意在遍历树节点的时候的深度问题：根节点深度为 0，访问左节点深度加一，访问右节点深度不变，然后根据传入的深度参数打印对应的空格数即可。

此外，在打印语法分析树的时候，对于 INT 类型的树节点，我们需要关注其节点的 name 值，如果为 HEX 或者 OCT，那么我们需要按照十六进制和八进制转换十进制的规则对于输出值进行转换。

在完成了对于语法分析树结构的定义和操作实现之后，我们就需要将相应的节点生成操作插入到 LR 语法分析的末尾，然后我们还需要谨慎的定义 error 的插入位置，我们需要尽可能的将 error 插入在较为底层的 LR 语法中，防止其将某些错误情况合并，造成错误的延迟发现。同时，为了使得错误分析更为清晰，我们重写了报错函数。

```
Stmt: Exp SEMI                                { $$ = newNode(@$.first_line, NOT_A_TOKEN, "Stmt", "", 2, $1, $2); }
    | CompSt                                  { $$ = newNode(@$.first_line, NOT_A_TOKEN, "Stmt", "", 1, $1); }
    | RETURN Exp SEMI                         { $$ = newNode(@$.first_line, NOT_A_TOKEN, "Stmt", "", 3, $1, $2, $3); }
    | IF LP Exp RP Stmt %prec LOWER_THAN_ELSE { $$ = newNode(@$.first_line, NOT_A_TOKEN, "Stmt", "", 5, $1, $2, $3, $4, $5); }
    | IF LP Exp RP Stmt ELSE Stmt             { $$ = newNode(@$.first_line, NOT_A_TOKEN, "Stmt", "", 7, $1, $2, $3, $4, $5, $6, $7); }
    | WHILE LP Exp RP Stmt                    { $$ = newNode(@$.first_line, NOT_A_TOKEN, "Stmt", "", 5, $1, $2, $3, $4, $5); }
    | error SEMI                              { synError = TRUE; }
    ;
```

图 3: 语义操作

```
yyerror(char* msg){
    fprintf(stderr,"error type B at line %d: %s.\n",yylineno, msg);
}
```

图 4: 错误函数

3 主程序模块

为了方便编译和调试，我们将 main 函数放在了 main.c 文件中，其具体逻辑入下：首先，读入我们的 cmm 文件，调用 yyparse() 函数对其进行词法分析和语法分析，如果此过程没有错误，则运行打印语法分析树的模块将程序的树结构完整输出到屏幕上，如果词法分析阶段或者是语法分析阶段检测到错误，那么我们就调用我们自己重写的 error 函数进行报错。

4 编译执行方式

程序的正确编译执行需要以下六个文件：enum.h, PTnode.h, lexical.l, main.c, syntax.y, makefile，只需要确保他们处在同一个目录下，然后执行 make 命令调用 makefile 文件执行自动编译即可。