编译原理实验一学习笔记

词法规则

1. 关键字、运算符和标点符号：对于关键字、运算符和标点符号，我们无须声明词法规则，只需在语法规则中直接使用单引号将他们括起来即可，比如 'while'、'+'。

2. 标识符：一个基本的标识符就是一个由大小写字母组成的字符序列。需要注意的是，下面的ID规则也能够匹配关键字（比如‘while’）等，上章中我们查看了Parser代码，知道ANTLR是如何处理这种歧义性的——选择所有匹配的备选分支中的第一条。因此，ID标识符应该放在关键字等定义之后。

```
// 匹配一个或者多个大小写字母
ID : [a-zA-Z]+;
```

3. 整数：整数是包括正数和负数的不以零开头的数字。

```
// 匹配一个整数
INTEGER : '-'?[1-9][0-9]*
    | '0'
    ;
```

4. 注释和空白字符：对于注释和空白字符，大多数情况下对于语法分析器是无用的（Python是一个例外，它的换行符表示一条命令的终止，特定数量的缩进指明嵌套的层级），因此我们可以使用ANTLR的skip指令来通知词法分析器将它们丢弃。

```
// 单行注释(以//开头，换行结束)
LINE_COMMENT : '//' .*? '\r'?'\n' -> skip;
// 多行注释(/* */包裹的所有字符)
COMMENT : '/*' .*? '*/' -> skip;
```

词法分析器可以接受许多 -> 操作符之后的指令，skip只是其中之一。例如，如果我们需要在语法分析器中对注释做一定处理，我们可以使用channel指令将某些词法符号送入一个“隐藏的通道”并输送给语法分析器。

大多数编程语言将空白符看成是词法符号间的分隔符，并将他们忽略。

```
// 匹配一个或者多个空白字符并将他们丢弃
WS : [ \t\r\n]+ -> skip;
```

语法规则

语法（grammar）包含了一系列描述语言结构的规则。这些规则不仅包括描述语法结构的规则，也包括描述标识符和整数之类的词汇符号（词法符号Token）的规则，即包含词法规则和语法规则。

注意：语法分析器的规则必须以小写字母开头，词法分析器的规则必须以大写字母开头。

1. 语法文件声明

语法由一个为该语法命名的头部定义和一系列可以互相引用的语言规则组成。grammar关键字用于语法文件命名，需要注意的是，命名须与文件名一致。

2. 语法导入

前两章的例子中，我们都是将词法规则和语法规则放在一个语法文件中，然而一个优雅的写法是将词法规则和语法规则进行拆分。lexer grammar关键字用于声明一个词法规则文件。如下是一个通用的词法规则文件定义。

```
// 通用的词法规则，注意是 lexer grammar
lexer grammar CommonLexerRules;
// 匹配标识符(+表示匹配一次或者多次)
ID : [a-zA-Z]+;
// 匹配整数
INT : [0-9]+;
// 匹配换行符(?表示匹配零次或者一次)
NEWLINE : '\r'?'\n';
// 丢弃空白字符
WS : [ \t]+ -> skip;
```

然后我们只需要import关键字，就可以轻松的将词法规则进行导入。如下是一个计算器的语法文件。

```
grammar LibExpr;
// 引入 CommonLexerRules.g4 中全部的词法规则
import CommonLexerRules;

prog : stat+;
stat : expr NEWLINE          # printExpr
   | ID '=' expr NEWLINE     # assign
   | NEWLINE                 # blank
   ;
expr : expr op=('*' | '/') expr    # MulDiv
   | expr op=('+' | '-') expr     # AddSub
   | INT                  # int
   | ID                   # id
   | '(' expr ')'           # parens
   | 'clear'                # clear
   ;

// 为上诉语法中使用的算术符命名
MUL : '*';
DIV : '/';
ADD : '+';
SUB : '-';
```

3. 备选分支命名（标签）

如果备选分支上面没有标签，ANTLR就只会为每条规则生成一个方法（监听器和访问器中的方法，用于对不同的输入进行不同的操作）。为备选分支添加一个标签，我们只需要在备选分支的右侧，以 # 开头，后面跟上任意的标识符即可。如上所示。需要注意的是，为一个规则的备选分支添加标签，要么全部添加，要么全部不添加。

4. 优先级

在第二章中我们讲述了ANTLR是如何处理歧义性语句（二义性文法）的：选择所有匹配的备选分支中的第一条。即ANTLR通过优先选择位置靠前的备选分支来解决歧义性问题，这也隐式地允许我们指定运算符优先级。例如，在上诉的例子中，乘除的优先级会比加减高。因此，ANTLR在解决1+2*3的歧义问题时，会优先处理乘法。

5. 结合性

默认情况下，ANTLR是左结合的，即将运算符从左到右地进行结合。但是有些情况下，比如指数运算符是从右向左结合的。1^2^3应该是3^(2^1)而不是(3^2)^1。我们可以使用assoc来手动指定结合性。

```
expr : expr '^' <assoc=right> expr    // ^ 是右结合的
    | INT
    ;
```

注意，在ANTLR4.2之后，<assoc=right>需要放在备选分支的最左侧，否则会收到警告。

```
expr : <assoc=right> expr '^' expr    // ^ 是右结合的
    | INT
    ;
```

Step 2

构造词法规则：

```
lexer grammar MIDLLexerRules;

LETTER : [a-z] | [A- Z];

DIGIT : [0-9]

UNDERLINE : _;

ID : LETTER (UNDERLINE?( LETTER | DIGIT))*;

INTEGER_TYPE_SUFFIX : l | L;

INTEGER : (0 | [1-9] [0-9]*) INTEGER_TYPE_SUFFIX?;

EXPONENT : ( e | E) ( + | - )? [0-9]+;

FLOAT_TYPE_SUFFIX : f | F | d | D;

FLOATING_PT:  [0-9]+ . [0-9]*  EXPONENT?  FLOAT_TYPE_SUFFIX?
            |  . [0-9]+  EXPONENT?  FLOAT_TYPE_SUFFIX?
            |  [0-9]+  EXPONENT  FLOAT_TYPE_SUFFIX?
            |  [0-9]+  EXPONENT?  FLOAT_TYPE_SUFFIX
            ;

ESCAPE_SEQUENCE : \ ( b | t | n | f | r | " | ' | \ );

CHAR : '(ESCAPE_SEQUENCE |  (~\ | ~') )';

STRING : " (ESCAPE_SEQUENCE |  (~\ | ~") )* " ;

BOOLEAN : TRUE | true | FALSE | false;
```

构造语法规则：

```
grammar MIDLGrammarRules;
import MIDLLexerRules;

specification : definition+;
```

definiton :
definiton -> type_decl “;” | module “;”
module -> “module” ID “{” definition { definition } “}”
type_decl -> struct_type | “struct” ID
struct_type-> “struct” ID “{” member_list “}”
member_list-> { type_spec declarators “;” }
type_spec -> scoped_name | base_type_spec | struct_type
scoped_name -> [“::”] ID { “::” ID }
base_type_spec->floating_pt_type|integer_type| “char” | “string” |
“boolean”
floating_pt_type -> “float” | “double” | “long double”
integer_type -> signed_int | unsigned_int
signed_int->(“short” | “int16”)
|(“long” | “int32”)
|(“long” “long” | “int64”)
| “int8”
unsigned_int -> (“unsigned” “short” | “uint16”)
| (“unsigned” “long” | “uint32”)
| (“unsigned” “long” “long” | “uint64”)
| “uint8”
declarators -> declarator { “,” declarator }
declarator -> simple_declarator | array_declarator
simple_declarator -> ID [“=” or_expr]
array_declarator -> ID “[” or_expr “]” [“=” exp_list]
exp_list -> “[” or_expr { “,” or_expr } “]”
or_expr -> xor_expr { “|” xor_expr }
xor_expr -> and_expr { “^” and_expr }
and_expr -> shift_expr { “&” shift_expr }
shift_expr -> add_expr { (“>>” | “<<”) add_expr }
add_expr -> mult_expr { (“+” | “-”) mult_expr }
mult_expr -> unary_expr { (“*” | “/” | “%”) unary_expr }
unary_expr -> [“-” | “+” | “~”] literal
literal -> INTEGER | FLOATING_PT | CHAR | STRING | BOOLEAN