



02 | 词法分析：识别Token也可以很简单吗？

2021-08-11 宫文学

《手把手带你写一门编程语言》

[课程介绍 >](#)



讲述：宫文学

时长 20:26 大小 18.72M



你好，我是宫文学。

上一节课，我们用了很简单的方法就实现了语法分析。但当时，我们省略了词法分析的任务，使用了一个 Token 的列表作为语法分析阶段的输入，而这个 Token 列表呢，就是词法分析的结果。

其实，编译器的第一项工作就是词法分析，也是你实现一门计算机语言的头一项基本功。今天，我们就来补补课，学习一下怎么实现词法分析功能，词法分析也就是把程序从字符串转换成 Token 串的过程。



词法分析难不难呢？我们来对比一下，语法分析的结果是一棵 AST 树，而词法分析的结果是一个列表。直观上看，列表就要比树结构简单一些，所以你大概会猜想到，词法分析应

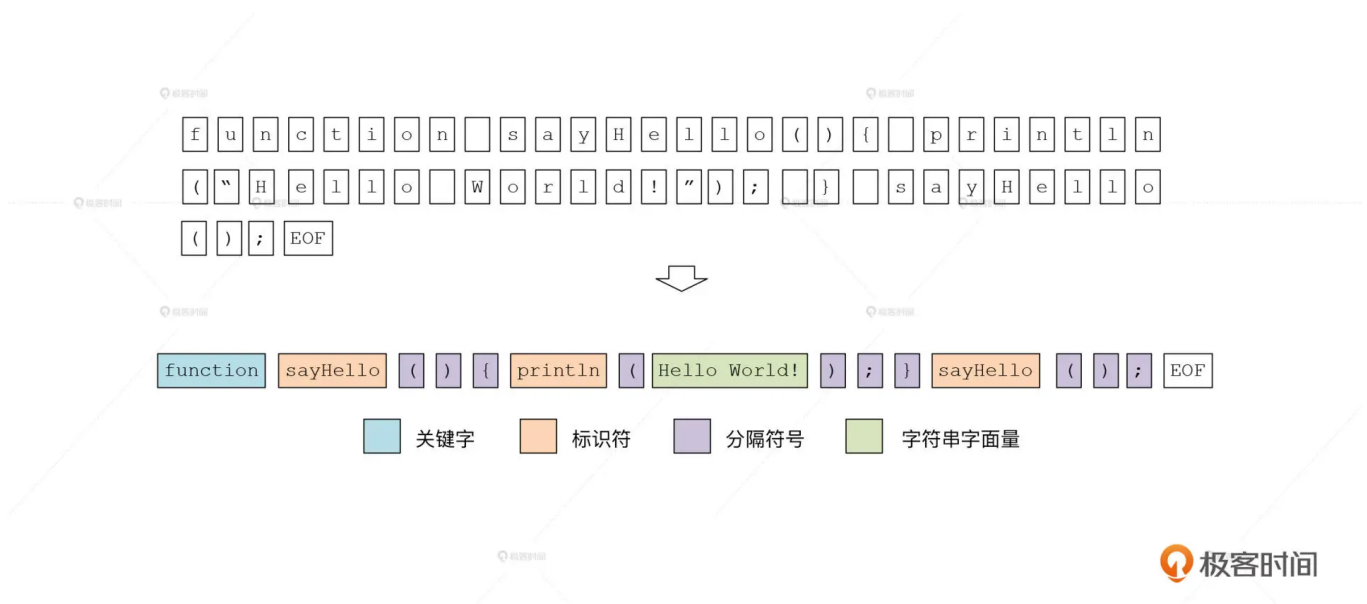
该会更简单一些。

那么，具体来说，**词法分析要用什么算法呢？词法是不是也像语法一样有规则？词法规则又是如何表达的？**这一节课，我会带着你实现一个词法分析器，来帮你掌握这些技能。

在这里，我有个好消息告诉你。你在上一节课学到的语法分析的技能，很多可以用在词法分析中，这会大大降低你的学习难度。好了，我们开始了。

词法分析的任务

你已经知道，词法分析的任务就是把程序从字符串转变成 Token 串，那它该怎么实现呢？我们这里先不讲具体的算法，先来看看下面这张示意图，分析一下，我们人类的大脑是如何把这个字符串断成一个个 Token 的？



你可能首先会想到，借助字符串中的空白字符（包括空格、回车、换行），把这个字符串截成一段段的，每一段作为一个 Token，行不行？

按照这个方法，function 关键字可以被单独识别出来。但是你看，我们还有一些圆括号、花括号等等，这些符号跟前一个单词之间并没有空格或回车，我们怎么把它们断开呢？

OK，你可以说，凡是遇到圆括号、花括号、加号、减号、点号等这些符号，我们把它们单独作为 Token 识别出来就好了。比如，对于 cat.weight 这样的对象属性访问的场景，点符号就是一个单独的 Token。

但是，你马上会发现这个规则仍然不能处理所有的情况，例如，对于一个浮点数的字面量“3.14”的情况，点符号是浮点数的一部分，不能作为单独的Token。我稍微解释一下，这里的字面量（Literal），是指写在程序中的常量值，包括整数值、浮点数值、字符串等。

此外，还有一些难处理的情况，比如像“==”、“+=”、“-=”、“-”、“&&”这些由两个或两个以上字符构成的运算符，程序处理时是要跟“=”、“+”、“-”等区分开的。

再比如，在JavaScript/TypeScript中，十六进制的字面量用“0x”开头，里面有a到f的字母，比如0x1F4；八进制的字面量用“0”开头，后面跟0~7的数字；而二进制的字面量用“0b”开头，后面跟着0或1。

所以，你可以看到，做词法分析需要考虑的情况还挺多，不是用简单的一两个规则就能解决的，我们必须寻找一种系统性的解决方法。

在这里，为了让你对词法分析的任务有更全面的了解，我梳理了各种不同的处理工作，你可以看看下面这张表：

分类	内容
关键字	break, if, do, instanceof, for, while, this, class, super等
标识符	如sayHello, foo, bar等
分隔符	(){}[];,.
运算符	> < ! ~ ? : == === >= <= != && ++ -- + - * / & ^ % += -= /= %= <<= >>= >>>= ==等
字面量	二进制、八进制、十六进制字面量 浮点数字面量 字符串字面量 正则表达式字面量
注释	单行注释、多行注释、Html注释、CData注释，这些不形成Token，不进入语法分析。
空白字符	空格、回车、Tab等，这些不形成Token。

那么，如何用系统性的方法进行词法分析呢？

借助这节课一开头的提示，我们试一下能否用语法分析的方法来处理词法，也就是说**像做语法分析一样，我们要先用一个规则来描述每个 Token 的词法，然后让程序基于词法规则来做处理。**

词法规则

同语法规则一样，我们可以用正则表达式来描述词法。在这里，标识符的规则是用字母开头，后面的字符可以是字母、数字或下划线，标识符的词法规则可以写成下面这样：

```
1 Identifier: [a-zA-Z_][a-zA-Z0-9_]* ;
```

[复制代码](#)

实际上，JavaScript 的标识符是允许使用合法的 Unicode 字符的，我们这里做了简化。

看上去，词法规则跟上一节学过的语法规则没什么不同嘛，只不过词法的构成要素是字符，而语法的构成要素是 Token。

表面上是这样，其实这里还是有一点不同的。实际上，词法规则使用的是正则文法（Formal Grammar），而语法规则使用的是上下文无关文法（Context-free Grammar，CFG）。正则文法是上下文无关文法的一个子集，至于对这两者差别的深入分析，我们还是放到后面的课上，这里我们先专注于完成词法分析功能。

我们再写一下前面讨论过的浮点数字面量的词法规则，这里同样是精简的版本，省略了指数部分、二进制、八进制以及十六进制的内容：

```
1 DecimalLiteral: IntegerLiteral '.' [0-9]*
2                | '.' [0-9]+
3                | IntegerLiteral
4                ;
5 IntegerLiteral: '0' | [1-9] [0-9]* ;
```

[复制代码](#)

对于上面这个 DecimalLiteral 词法规则，我们总结一下有这几个特点：

一个合法的浮点数可以是好几种格式，3.14、.14、3 等等都行；

整数部分要么只有一个 0，要么是 1~9 开头的数字；

可以没有小数点前的整数部分，但这时候小数点后至少要有一位数字，否则这就只剩了一个点号了；

也可以完全没有小数部分，只有整数部分。

好了，目前我们已经知道如何用词法规则来描述不同的 Token 了。接下来，我们要做的就是用程序实现这些词法规则，然后生成 Token。


用程序实现词法分析

上节课我们在讲语法分析的时候，提到了递归下降算法。这个算法比较让人喜欢的一点是，程序结构基本上就是对语法规则的一对一翻译。

其实词法分析程序也是一样的，比如我们要识别一个浮点数，我们照样可以根据上述 `DecimalLiteral` 的几条规则一条条地匹配过去。

首先，我们匹配第一条规则，就是既有整数部分又有小数部分的情况；如果匹配不上，就尝试第二条规则，也就是以小数点开头的情况；如果还匹配不上，就尝试第三条，即只有整数部分的情况。只要这三条匹配里有一条成功，就意味着我们匹配浮点数成功。

我们来看看具体的程序实现：

 复制代码

```
1  if (this.isDigit(ch)){
2      this.stream.next();
3      let ch1 = this.stream.peek();
4      let literal:string = '';
5      //首先解析整数部分
6      if(ch == '0'){//暂不支持八进制、二进制、十六进制
7          if (!(ch1>='1' && ch1<='9')){
8              literal = '0'; //整数部分只有0
9          }
10         else {报编译错误}
11     }
12     else if(ch>='1' && ch<='9'){
13         literal += ch;
14         while(this.isDigit(ch1)){
15             ch = this.stream.next();
16             literal += ch;
17             ch1 = this.stream.peek();
```

```
18     }
19 }
20 //解析小数部分
21 if (ch1 == '.'){
22     literal += '.';
23     this.stream.next();
24     ch1 = this.stream.peek();
25     while(this.isDigit(ch1)){
26         ch = this.stream.next();
27         literal += ch;
28         ch1 = this.stream.peek();
29     }
30     return {kind:TokenKind.DecimalLiteral, text:literal};
31 }
32 else{
33     //返回一个整型直度量
34     return {kind:TokenKind.IntegerLiteral, text:literal};
35 }
36 }
37 //解析以.开头的小数，要求后面至少有一个数字
38 else if (ch == '.'){
39     this.stream.next();
40     let ch1 = this.stream.peek();
41     if (this.isDigit(ch1)){
42         //小数字面量
43         let literal = '.';
44         while(this.isDigit(ch1)){
45             ch = this.stream.next();
46             literal += ch;
47             ch1 = this.stream.peek();
48         }
49         return {kind:TokenKind.DecimalLiteral, text:literal};
50     }
51     else{
52         ...
53     }
54 }
```

如果浮点数匹配不成功呢？也没关系。其实计算机语言的词法规则分为很多条，有匹配浮点数的，也有匹配标识符的，还有匹配运算符的，等等。我们可以把这些词法规则依次匹配过去。只要某个规则匹配成功了，我们就算识别出了一种 Token。

这么看来，**词法分析的过程，就是依次匹配不同的词法规则的过程**。匹配成功以后，就把这个 Token 从字符串中截下来，再去尝试匹配下一个 Token。

这里有两个细节你要注意一下：一是当扫描到注释的时候，直接去掉，并不生成 Token；第二是对于空白字符的处理，也是一样，直接去掉。所以整体的处理逻辑如下：

[复制代码](#)

```
1 while (!EOF){  
2     跳过空白字符；  
3     如果是注释，丢弃掉；  
4     尝试匹配词法规则1；  
5     不成功，则回溯，尝试匹配词法规则2；  
6     不成功，则回溯，尝试匹配词法规则3；  
7     ...  
8     直到成功匹配一个词法规则；  
9 }
```

你根据这个逻辑，就可以写出整个词法分析程序了！

不过，你可能很快就发现了一个问题，我们的词法规则很多，可能有几十条甚至上百条。如果每次都从头开始依次尝试，会造成很多浪费，词法分析的性能也会很低，那我们有没有什么办法提速呢？

接下来，我们就把算法优化一下。

有限自动机：提升词法分析性能

怎么优化呢？你可能马上会想到一个好办法：虽然词法规则很多，但大部分 Token 都可以通过开头的第一个字符区分开。

如果第一个字符是 - 号，那么只可能是三种 Token：-，-= 和 -；

如果第一个字符是一个字母，那么只可能是标识符或各种关键字；

如果第一个字符是 0，那么必然是十六进制字面量、八进制字面量或者二进制字面量。

你看，通过预读一个字符，程序可以马上缩小选择范围，使得性能大大提升。

尝到甜头以后，我们马上可以想到，如果需要的话，能不能再继续预读第二个字符，进一步确定是哪个 Token 呢？

当然是可以的。比如，对于`-`、`--`和`-`这三个Token，我们可以再往下预读一个字符。如果是`-`号，那么就收获一个`-`Token；如果是`=`，那么就收获一个`--`Token；除了这两种之外，后面不管是什么字符，我们都把`-`号单独作为一个Token提取出来。

我们把这个判断过程画成一个状态迁移图，这个状态迁移图我们叫做一个有限自动机。

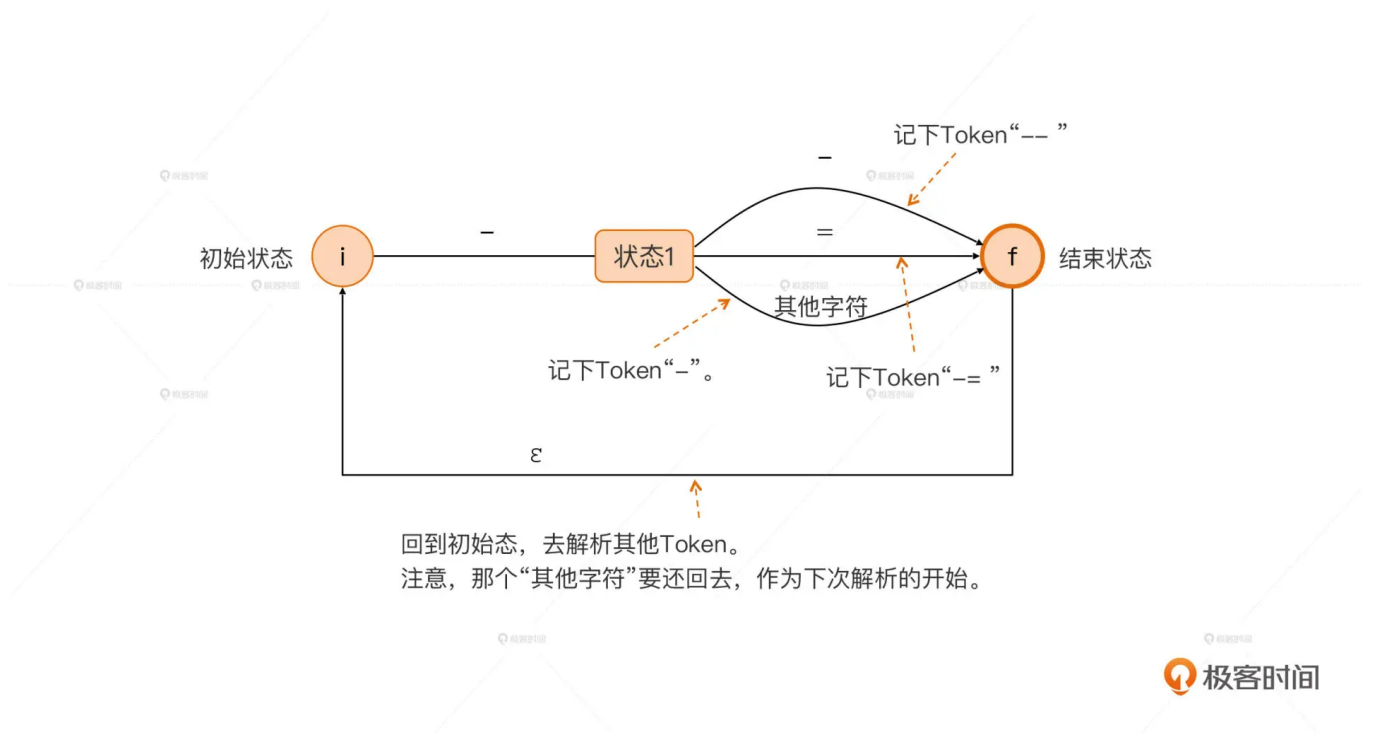


图2：解析`-`、`--`和`=`三种Token的有限自动机

恭喜你，到这里你已经在不知不觉间，实现出了教科书级的方法，也就是通过有限自动机（Finite-State Automata，FSA）来做词法分析了。

有限自动机可以分为两种。如果一个有限自动机针对每个输入，都会发生一次确定的迁移，这种有限自动机就被叫做确定性有限自动机（Deterministic Finite-State Automata，DFA）。

与之对应的，是另一种有限自动机，叫做非确定性的有限自动机（Non-deterministic Finite-State Automata，NFA）。它在某些状态下，针对一个输入可能会迁移到不止一个状态，或者在没有任何输入的情况下，也会从一个状态迁移到另一个状态。

不过，任何一个NFA，都可以通过一个算法转换成DFA，这个算法你可以参见[《编译原理之美》](#)中的算法篇。

言归正传，我们刚才已经画出了能识别三种 Token 的有限自动机，我们还可以把这个有限自动机扩大，像下图一样增加对多行注释、/、/=、标识符、关键字和空白字符的处理能力。

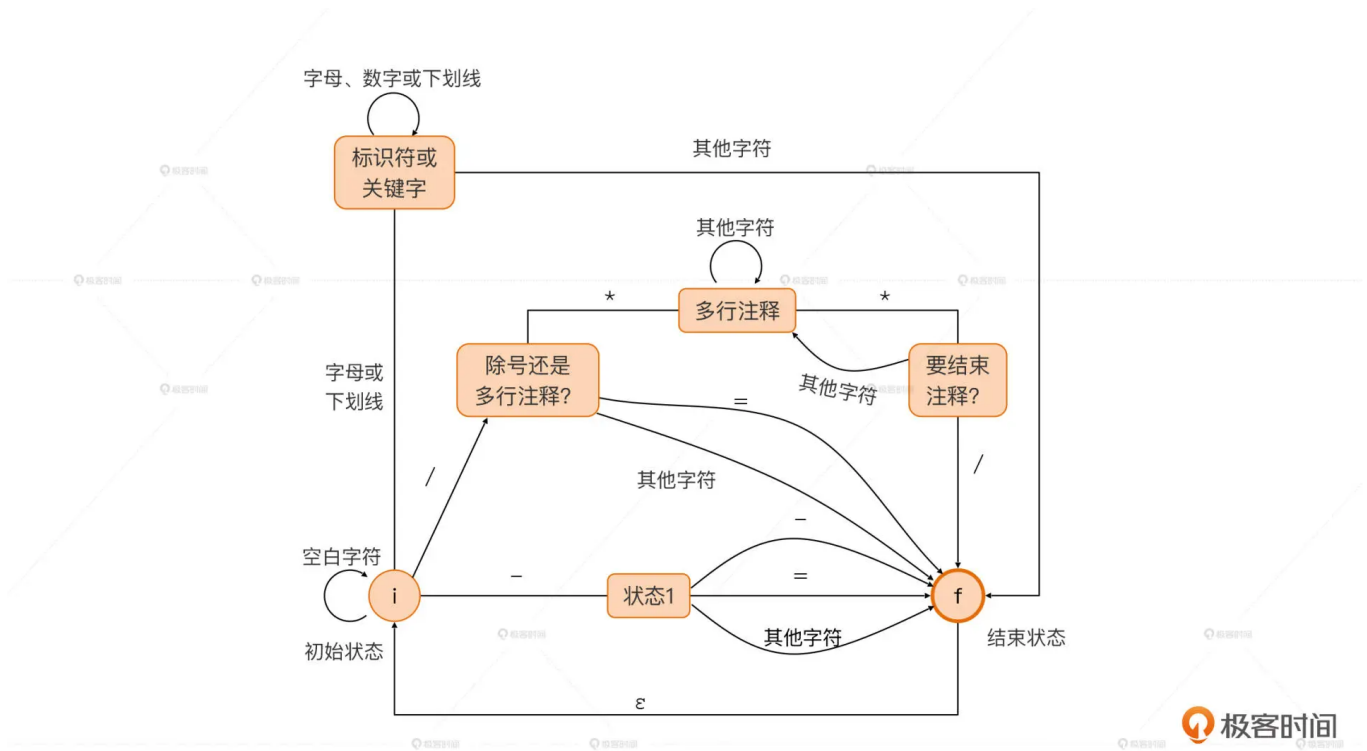


图3：增加了处理能力的有限自动机

你还可以继续扩展这个图，直到它能够提取所有的 Token 为止。有了这个大图之后，你就可以照着这个图写程序了。我这里给出了一个程序片段，供你参考。

复制代码

```
1 //从字符串流中获取一个新Token。
2 private getAToken():Token{
3     this.skipWhiteSpaces();
4     if (this.stream.eof()){
5         return {kind:TokenKind.EOF, text:""};
6     }
7     else{
8         //预读一个Token
9         let ch:string = this.stream.peek();
10        //处理标识符（包含关键字）
11        if (this.isLetter(ch) || ch == '_'){
12            return this.parseIdentifer();
13        }
14        //处理字符串字面量
15        else if (ch == '"'){
16            return this.parseStringLiteral();
17        }
18        else if (ch == '(' || ch == ')' || ch == '{' || ch == '}' || ch == '['
```

```
19         ch == ',' || ch == ';' || ch == ':' || ch == '?' || ch == '@'
20         this.stream.next();
21         return {kind:TokenKind.Separator,text:ch};
22     }
23     //处理多行注释、/、/=
24     else if (ch == '/') {
25         this.stream.next();
26         let ch1 = this.stream.peek();
27         if (ch1 == '*') {
28             this.skipMultipleLineComments();
29             return this.getAToken();
30         }
31         else if (ch1 == '/') {
32             this.skipSingleLineComment();
33             return this.getAToken();
34         }
35         else if (ch1 == '=') {
36             this.stream.next();
37             return {kind:TokenKind.Operator,text:'/= '};
38         }
39         else {
40             return {kind:TokenKind.Operator,text:'/ '};
41         }
42     }
43     //处理+、+=、++
44     else if (ch == '+') {
45         this.stream.next();
46         let ch1 = this.stream.peek();
47         if (ch1 == '+') {
48             this.stream.next();
49             return {kind:TokenKind.Operator,text:'++ '};
50         } else if (ch1 == '=') {
51             this.stream.next();
52             return {kind:TokenKind.Operator,text:'+= '};
53         }
54         else {
55             return {kind:TokenKind.Operator,text:'+'};
56         }
57     }
58     //省略了更多的情况.....
59     else {
60         //暂时去掉不能识别的字符
61         console.log("Unrecognized pattern meeting ': ' +ch+'', at" + this.
62             this.stream.next();
63             return this.getAToken();
64     }
65 }
66 }
```

如果你仔细看上面的图 3，你会发现，我的标识符和关键字都是统一处理的，因为标识符的词法其实已经包含了关键字。**做词法分析的一个最佳实践，就是先把标识符和关键字统一提取出来，然后再从里面把关键字单独提取出来就行了。**

好了，前面我们通过“预读”的思路，升级了词法分析的算法，使之能够正确地提取出所有的 Token。在这个过程中，我们已经不知不觉摸索出了编译原理教科书中都会讲的有限自动机算法。

可是这还没完。你可能会想到，采用有限自动机的情况下，程序每次都可以根据下一个输入，准确地确定接下来应该执行的逻辑，完全规避了回溯的问题。那我们是不是也可以用这个思路来提升语法分析的效率呢？好想法，我们现在就来试试看。


思路迁移：提升语法分析效率

我们先来回顾一下在语法分析中，回溯是怎么发生的，以及它会造成怎样的性能上的损失。

在语法分析的递归下降算法中，当一个语法规则由多个子规则构成时，我们采用的是挨个试的方法来进行匹配。比如，首先试一下某语句是否是一个函数声明，接着试一下它是否是一个函数调用。

在匹配某个语法规则失败的时候，我们需要把已经消化掉的 Token 还回去，恢复到没做匹配之前的状态，这个过程叫做回溯（Backtracking）。

由于整个算法是递归下降的，某一级的匹配失败也可能会导致上一级的失败，从而导致逐级回溯。比如，我们用下面的表达式的语法规则去匹配 $2+3*5$ 的时候，就会发生逐级回溯的现象：

 复制代码


```
1 //加法表达式，可以是一个乘法表达式，或者是一个乘法表达式+另一个加法表达式
2 add : mul
3     | mul '+' add
4     ;
5 //乘法表达式，可以是一个基础表达式，或者是一个基本表达式*另一个乘法表达式
6 mul : pri
7     | pri '*' mul
8     ;
9 //基础表达式，可以是一个整形字面量，或者是括号括起来的一个加法表达式
```

```

10 pri : IntLiteral
11     | '(' add ')'
12     ;

```

匹配过程是这样的：

 复制代码

```

1  2+3*5符合add规则吗？
2  ->2+3*5符合mul规则吗？
3  ->2+3*5符合pri规则吗？
4  ->2+3*5符合IntLiteral吗？不符合，因为它显然不只是一个整数常量
5  ->2+3*5符合 '(' add ')' 规则吗？不符合，因为第一个Token不是 '('
6  ->回溯到mul，去尝试第二个选项
7  ->2+3*5符合pri '*' mul吗？
8  ->匹配一个pri
9  ->匹配一个IntLiteral，成功
10 ->再匹配一个 '*', 失败，遇到的是 '+'
11 ->回溯到add，去尝试第二个选项
12 ->2+3*5符合mul '+' add吗？
13 ->匹配一个mul
14 ->匹配一个pri
15 ->匹配一个IntLiteral，成功
16 ->匹配一个 '+', 成功
17 ->匹配一个add
18 接下来，又回遇到一开头使用add规则的那种回溯的情形
19 ...

```

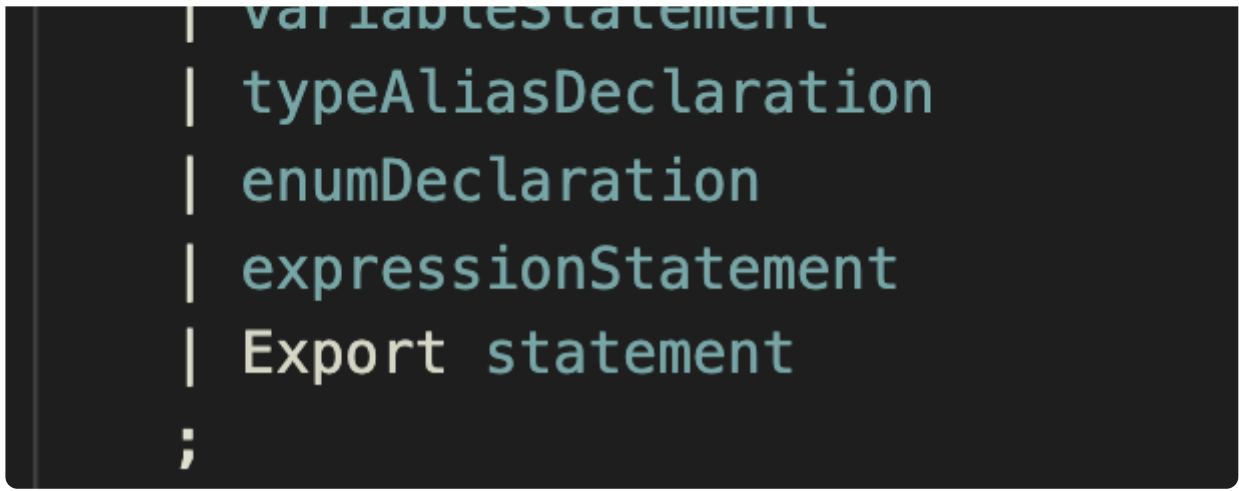
在这个例子中，我们还只是采用了 3 级嵌套的语法，如果语法的层次更多的话，由回溯造成的无用功就更多。从这里，你应该能体会到回溯的危害了。

不过，受到词法分析中预读字符的启发，我们在语法分析中也可以通过预读一个 Token 可以减少尝试的次数。比如，如果第一个 Token 是 “function” 关键字，那肯定是要匹配一个函数声明，这就不用考虑其他选项了，这不就可以提高性能了吗？

确实是这样的。在语法分析中，我们经常可以遇到有多个语法规则可供选择的情况。比如，在上一节的例子中，statement 就有两种可能性，这还是简化的情况；在完整的语法规则中，statement 有 20 多种可能性，像 if 语句、各种循环语句、各种声明等，都属于 statement。在这种情况下，预读所带来的性能改善就非常可观了！

statement

```
: block
| importStatement
| exportStatement
| emptyStatement
| abstractDeclaration
| decoratorList
| classDeclaration
| interfaceDeclaration
| namespaceDeclaration
| ifStatement
| iterationStatement
| continueStatement
| breakStatement
| returnStatement
| yieldStatement
| withStatement
| labelledStatement
| switchStatement
| throwStatement
| tryStatement
| debuggerStatement
| functionDeclaration
| arrowFunctionDeclaration
| generatorFunctionDeclaration
| variableStatement
```



为了通过预读实现性能的优化，我们需要知道每个语法规则开头的 Token 都可能有哪些。比如，if 语句永远是以 if 开头的，这个比较简单。但有的语法规则就比较复杂一些，比如表达式语句，它本身就是由很多子规则构成的，所以其第一个 Token 有很多可能：

[复制代码](#)

```
1 ++i;    //第一个Token是++
2 foo();  //第一个Token是一个标识符
3 "Hello World"; //第一个Token是一个字面量
4 (2+3);  //第一个Token是括号
```

这怎么办呢？也好办。我们把表达式语句所有可能出现的第一个 Token 的集合，叫做 First 集合。

对于 statement 来说，我们可以求出每个子规则的 First 集合，只要所有的这些集合都没有交集，那么我们总是可以通过预读一个 Token 来决定采用哪条子规则的。

如果整个语法规则都只需要预读一个 Token 就可以实现，那么我们把这个语法规则叫做 LL(1) 文法，而我们刚才说到的通过预读来唯一确定分支路径的算法，就是 LL(1) 算法，这个算法避免了由于回溯而导致的性能开销。

不过，对于 LL 两个字母的含义，我们留到下节课再讲，你现在只要记住这种算法名称叫做 LL 算法就好了。LL(1) 中的 1，意思是需要预读一个 Token。

另外，LL 算法还有一种特殊情况需要处理，比如语句块中可以有一个语句列表（statementList），也可能是一个空语句块：

```
1 block
2     : '{' statementList? '}'
3     ;
4 statementList
5     : statement+
6     ;
```

我们在解析语句块的时候，取出了 “{” 以后，后面可能是一个语句列表（statementList），也有可能直接遇到 “}”，这就意味着这是一个空语句块，也就是 “{}”。

那这个时候呢，我们在程序里要多加一个判断：如果预读的 Token 在 statementList 的 First 集合里，那么就要去解析一个 statementList；而如果遇到的是 “}”，那就没必要了。

由于这个 “}” 是出现在 statemtnList 之后的，我们可以说 “}” 属于 statementList 的 Follow 集合。我们根据 statemntList 的 First 和 Follow 集合，就能解决语法规则中 statementList 后面跟一个? 号的情况。除了? 号，在处理 * 号和 + 号的时候，也都会需要用到 Follow 集合。

总结起来，完整的 LL 算法，需要用到每个语法规则的 First 和 Follow 集合，来确定我们应该采用哪个子规则的分支，从而避免回溯，实现性能上的提升。

课程小结

好了，这一节课就先到这里了。这节课，我们不仅学到了如何做词法分析，还把语法分析的技能又提升了一级。

总结一下，词法分析也可以使用 EBNF 来描述词法规则，并且也可以使用类似递归下降的算法。不过，通过预读字符，我们可以降低尝试的次数。最终，通过构造一个有限自动机，我们可以对每一个输入都做一个确定的状态转移，从而高效地实现词法分析。

接着，我们把“预读”的思维放到了语法分析中的递归下降算法上，又介绍了 First 集合和 Follow 集合的概念，借助这两个集合，我们不需要回溯，也能够实现语法分析，从而大大提高了语法分析的效率。

思考题

最后我给你出了一个思考题。你看，我们在程序里需要处理负数字面量，比如 -3。你认为，负数的负号应该是作为运算符来处理，分别解析成一个 - 号和一个字面量 3，还是把 -3 整体作为一个字面量识别出来呢？为什么？

欢迎你和我一起学习编程语言，也欢迎你将这门课分享给更多对编程语言感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

[🔗 这节课的示例代码在这里！](#)

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

👍 赞 0 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | 实现一门超简单的语言最快需要多久？

精选留言 (2)

💬 写留言



王

2021-08-11

老师，我觉得应该把-3整体当做一个字面量解析。如果解析成一个-和一个3的话，就是减法运算了，减法是需要两个数的。如果非要解析为减法运算，是不是被减数还得默认设置为0



springXu

2021-08-11

把-3作为整体，计算机中负数是用补码来表示负数的。这样把字符转换成数字方便了。



