

02 | 正则文法和有限自动机：纯手工打造词法分析器

2019-08-16 宫文学 来自北京

《编译原理之美》



上一讲，我提到词法分析的工作是将一个长长的字符串识别出一个个的单词，这一个个单词就是 Token。而且词法分析的工作是一边读取一边识别字符串的，不是把字符串都读到内存再识别。你在听一位朋友讲话的时候，其实也是同样的过程，一边听，一边提取信息。

那么问题来了，字符串是一连串字符形成的，怎么把它断开成一个个的 Token 呢？分割的依据是什么呢？本节课，我会通过讲解正则表达式（Regular Expression）和有限自动机的知识带你解决这个问题。

其实，我们手工打造词法分析器的过程，就是写出正则表达式，画出有限自动机的图形，然后根据图形直观地写出解析代码的过程。而我今天带你写的词法分析器，能够分析以下 3 个程序语句：

```
age >= 45
```

```
int age = 40
```

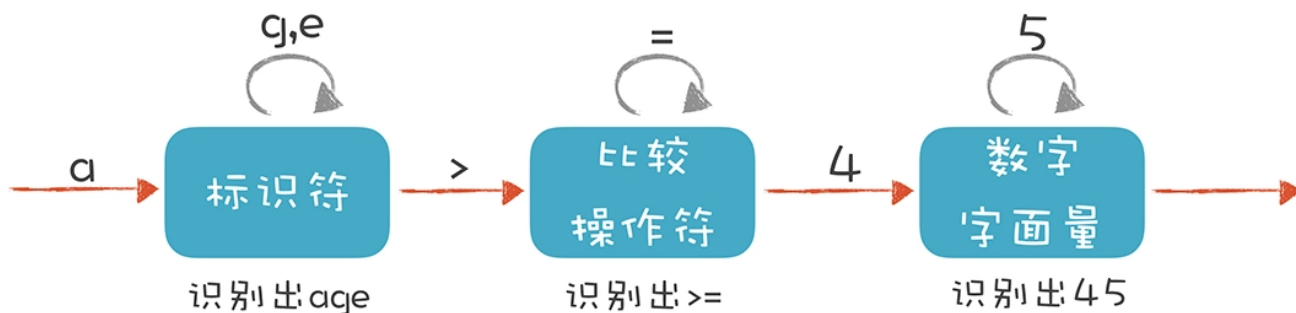
$$2+3*5$$

它们分别是关系表达式、变量声明和初始化语句，以及算术表达式。

接下来，我们先来解析一下 “age >= 45” 这个关系表达式，这样你就能理解有限自动机的概念，知道它是做词法解析的核心机制了。

解析 age >= 45

在 “[01 | 理解代码：编译器的前端技术](#)” 里，我举了一个词法分析的例子，并且提出词法分析要用到有限自动机。当时，我画了这样一个示意图：



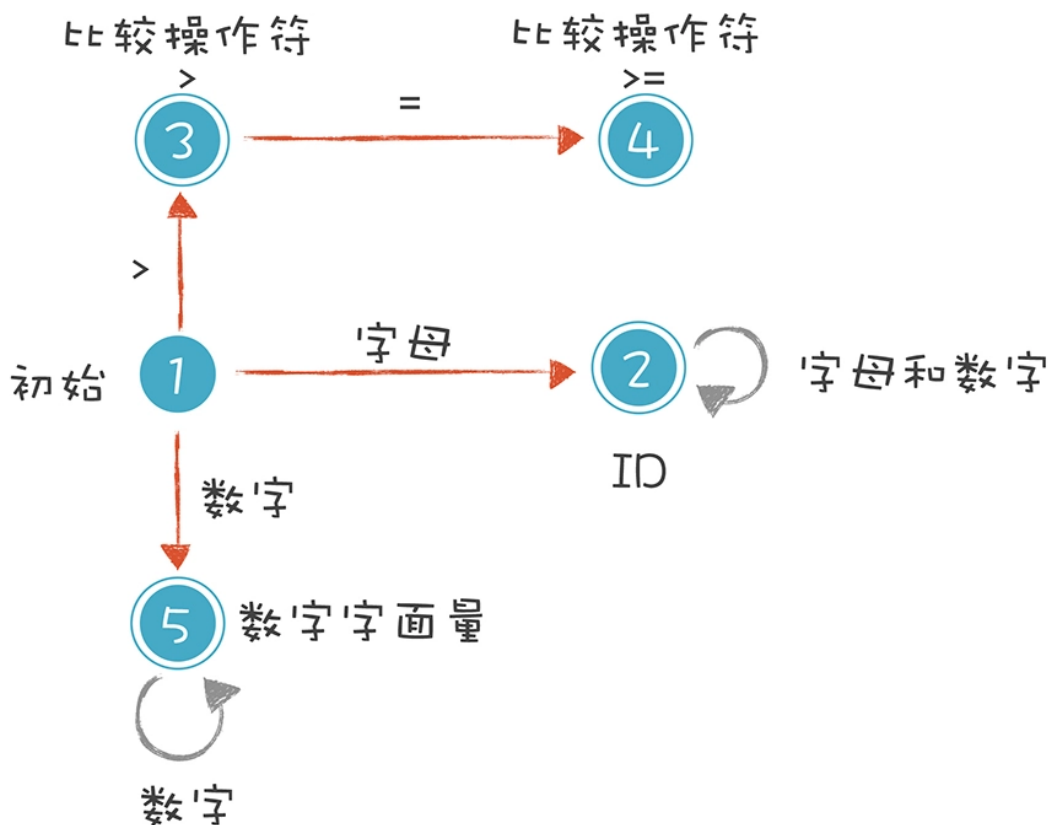
我们来描述一下标识符、比较操作符和数字字面量这三种 Token 的词法规则。

标识符：第一个字符必须是字母，后面的字符可以是字母或数字。

比较操作符：> 和 >=（其他比较操作符暂时忽略）。

数字字面量：全部由数字构成（像带小数点的浮点数，暂时不管它）。

我们就是依据这样的规则，来构造有限自动机的。这样，词法分析程序在遇到 age、>= 和 45 时，会分别识别成标识符、比较操作符和数字字面量。不过上面的图只是一个简化的示意图，一个严格意义上的有限自动机是下面这种画法：



我来解释一下上图的 5 种状态。

1. 初始状态：刚开始启动词法分析的时候，程序所处的状态。

2. 标识符状态：在初始状态时，当第一个字符是字母的时候，迁移到状态 2。当后续字符是字母和数字时，保留在状态 2。如果不是，就离开状态 2，写下该 Token，回到初始状态。


3. 大于操作符 (GT)：在初始状态时，当第一个字符是 > 时，进入这个状态。它是比较操作符的一种情况。

4. 大于等于操作符 (GE)：如果状态 3 的下一个字符是 =，就进入状态 4，变成 >=。它也是比较操作符的一种情况。

5. 数字字面量：在初始状态时，下一个字符是数字，进入这个状态。如果后续仍是数字，就保持在状态 5。

这里我想补充一下，你能看到上图中的圆圈有单线的也有双线的。双线的意思是这个状态已经是一个合法的 Token 了，单线的意思是这个状态还是临时状态。

按照这 5 种状态迁移过程，你很容易编成程序（我用 Java 写了代码示例，你可以用自己熟悉的语言编写）。我们先从状态 1 开始，在遇到不同的字符时，分别进入 2、3、5 三个状态：


 复制代码

```
1 DfaState newState = DfaState.Initial;
2 if (isAlpha(ch)) {                //第一个字符是字母
3     newState = DfaState.Id; //进入Id状态
4     token.type = TokenType.Identifier;
5     tokenText.append(ch);
6 } else if (isDigit(ch)) {         //第一个字符是数字
7     newState = DfaState.IntLiteral;
8     token.type = TokenType.IntLiteral;
9     tokenText.append(ch);
10 } else if (ch == '>') {           //第一个字符是>
11     newState = DfaState.GT;
12     token.type = TokenType.GT;
13     tokenText.append(ch);
14 }
```

上面的代码中，我用 Java 中的枚举（enum）类型定义了一些枚举值来代表不同的状态，让代码更容易读。

其中 Token 是自定义的一个数据结构，它有两个主要的属性：一个是“type”，就是 Token 的类型，它用的也是一个枚举类型的值；一个是“text”，也就是这个 Token 的文本值。

我们接着处理进入 2、3、5 三个状态之后的状态迁移过程：

 复制代码

```
1 case Initial:
2     state = initToken(ch);          //重新确定后续状态
3     break;
4 case Id:
5     if (isAlpha(ch) || isDigit(ch)) {
6         tokenText.append(ch);        //保持标识符状态
7     } else {
8         state = initToken(ch); //退出标识符状态，并保存Token
```

```

9      }
10     break;
11 case GT:
12     if (ch == '=') {
13         token.type = TokenType.GE; //转换成GE
14         state = DfaState.GE;
15         tokenText.append(ch);
16     } else {
17         state = initToken(ch); //退出GT状态, 并保存Token
18     }
19     break;
20 case GE:
21     state = initToken(ch); //退出当前状态, 并保存Token
22     break;
23 case IntLiteral:
24     if (isDigit(ch)) {
25         tokenText.append(ch); //继续保持在数字字面量状态
26     } else {
27         state = initToken(ch); //退出当前状态, 并保存Token
28     }
29     break;

```

运行这个示例程序，你就会成功地解析类似 “age >= 45” 这样的程序语句。不过，你可以先根据我的讲解自己实现一下，然后再去参考这个示例程序。

示例程序的输出如下，其中第一列是 Token 的类型，第二列是 Token 的文本值：

 复制代码

```

1 Identifier  age
2 GE         >=
3 IntLiteral 45

```

上面的例子虽然简单，但其实已经讲清楚了词法原理，**就是依据构造好的有限自动机，在不同的状态中迁移，从而解析出 Token 来**。你只要再扩展这个有限自动机，增加里面的状态和迁移路线，就可以逐步实现一个完整的词法分析器了。

初识正则表达式

但是，这里存在一个问题。我们在描述词法规则时用了自然语言。比如，在描述标识符的规则时，我们是这样表达的：

第一个字符必须是字母，后面的字符可以是字母或数字。

这样描述规则并不精确，我们需要换一种严谨的表达方式，这种方式就是**正则表达式**。

上面的例子涉及了 4 种 Token，这 4 种 Token 用正则表达式表达，是下面的样子：

复制代码

```
1 Id :      [a-zA-Z_] ([a-zA-Z_] | [0-9]) *
2 IntLiteral: [0-9] +
3 GT :      '>'
4 GE :      '>='
```

我先来解释一下这几个规则中用到的一些符号：

[]	代表匹配方括号中的字符之一
[abc]	匹配a或b或c
[a-z]	匹配从a到z的任何一个字符
*	匹配0个或者多个前面的元素
+	匹配1个或者多个前面的元素
"	严格匹配单引号里面的所有字符，如：'if'或者'int'

需要注意的是，不同语言的标识符、整型字面量的规则可能是不同的。比如，有的语言可以允许用 Unicode 作为标识符，也就是说变量名称可以是中文的。还有的语言规定，十进制数字字面量的第一位不能是 0。这时候正则表达式会有不同的写法，对应的有限自动机自然也不同。而且，不同工具的正则表达式写法会略有不同，但大致是差不多的。


我在本节课讲正则表达式，主要是为了让词法规则更为严谨，当然了，也是为后面的内容做铺垫。在后面的课程中，我会带你用工具生成词法分析器，而工具读取的就是用正则表达式描述的词法规则。到时候，我们会把所有常用的词法都用正则表达式描述出来。

不过在这之前，如果你想主动了解更完整的正则表达式规则，完全可以参考自己所采用的正则表达式工具的文档。比如，Java 的正则式表达式工具在 `java.util.regex` 包中，在其 Javadoc 中有详细的规则说明。

解析 `int age = 40`，处理标识符和关键字规则的冲突

说完正则表达式，我们接着去处理其他词法，比如解析 “`int age = 40`” 这个语句，以这个语句为例研究一下词法分析中会遇到的问题：多个规则之间的冲突。

如果我们把这个语句涉及的词法规则用正则表达式写出来，是下面这个样子：

 复制代码

```
1 Int:           'int'
2 Id :           [a-zA-Z_] ([a-zA-Z_] | [0-9]) *
3 Assignment :   '='
```

这时候，你可能会发现这样一个问题：`int` 这个关键字，与标识符很相似，都是以字母开头，后面跟着其他字母。

换句话说，`int` 这个字符串，既符合标识符的规则，又符合 `int` 这个关键字的规则，这两个规则发生了重叠。这样就起冲突了，我们扫描字符串的时候，到底该用哪个规则呢？

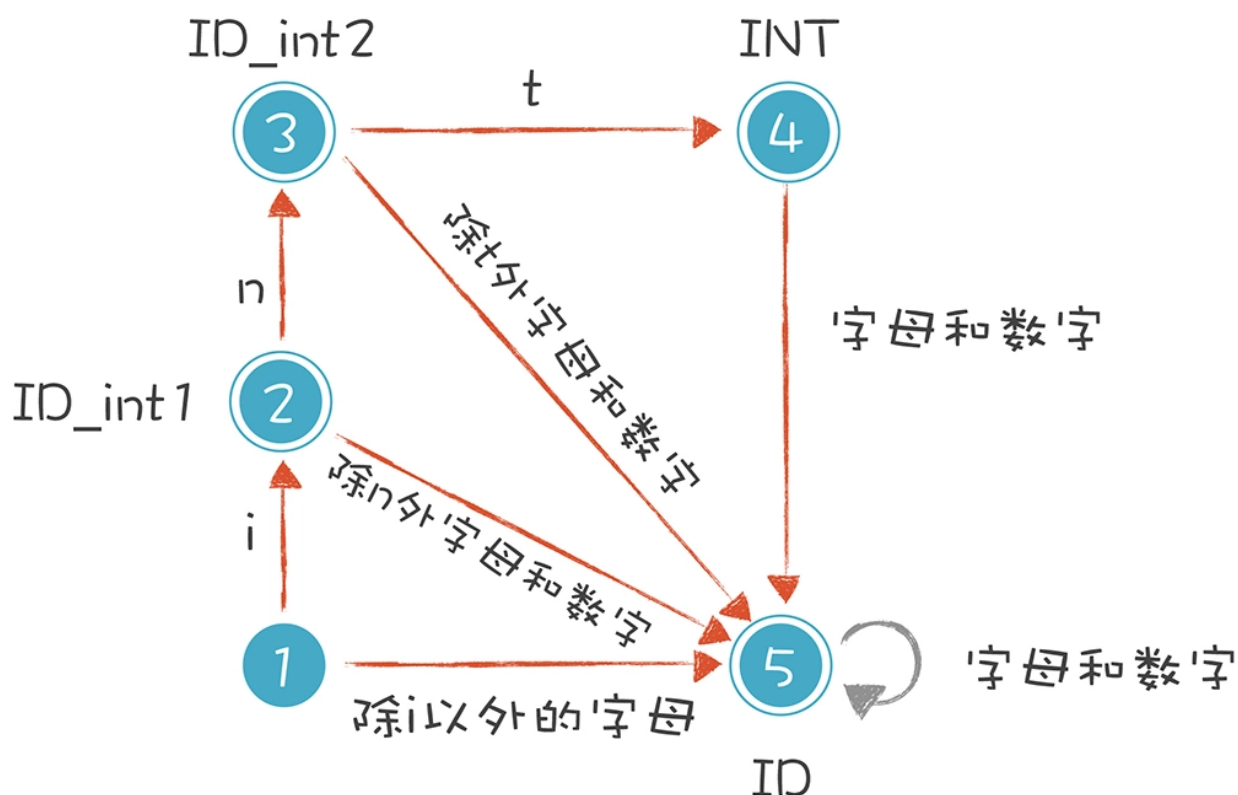
当然，我们心里知道，`int` 这个关键字的规则，比标识符的规则优先级高。普通的标识符是不允许跟这些关键字重名的。

在这里，我们来回顾一下：什么是关键字？

关键字是语言设计中作为语法要素的词汇，例如表示数据类型的 `int`、`char`，表示程序结构的 `while`、`if`，表述特殊数据取值的 `null`、`NAN` 等。

除了关键字，还有一些词汇叫保留字。保留字在当前的语言设计中还没用到，但是保留下来，因为将来会用到。我们命名自己的变量、类名称，不可以用到跟关键字和保留字相同的字符串。**那么我们在词法分析器中，如何把关键字和保留字跟标识符区分开呢？**

以 “int age = 40” 为例，我们把有限自动机修改成下面的样子，借此解决关键字和标识符的冲突。



这个思路其实很简单。在识别普通的标识符之前，你先看看它是关键字还是保留字就可以了。具体做法是：

当第一个字符是 `i` 的时候，我们让它进入一个特殊的状态。接下来，如果它遇到 `n` 和 `t`，就进入状态 4。但这还没有结束，如果后续的字符还有其他的字母和数字，它又变成了普通的标识符。比如，我们可以声明一个 `intA` (`int` 和 `A` 是连着的) 这样的变量，而不会跟 `int` 关键字冲突。

相应的代码也修改一下，文稿里的第一段代码要改成：

[📄 复制代码](#)

```
1 if (isAlpha(ch)) {
2     if (ch == 'i') {
3         newState = DfaState.Id_int1; //对字符i特殊处理
4     } else {
5         newState = DfaState.Id;
6     }
7     ... //后续代码
8 }
```


第二段代码要增加下面的语句：

[📄 复制代码](#)

```
1 case Id_int1:
2     if (ch == 'n') {
3         state = DfaState.Id_int2;
4         tokenText.append(ch);
5     }
6     else if (isDigit(ch) || isAlpha(ch)){
7         state = DfaState.Id; //切换回Id状态
8         tokenText.append(ch);
9     }
10    else {
11        state = initToken(ch);
12    }
13    break;
14 case Id_int2:
15     if (ch == 't') {
16         state = DfaState.Id_int3;
17         tokenText.append(ch);
18     }
19     else if (isDigit(ch) || isAlpha(ch)){
20         state = DfaState.Id; //切换回Id状态
21         tokenText.append(ch);
22     }
23     else {
24         state = initToken(ch);
25     }
26    break;
27 case Id_int3:
28     if (isBlank(ch)) {
29         token.type = TokenType.Int;
30         state = initToken(ch);
31     }
32     else{
33         state = DfaState.Id; //切换回Id状态
```


```
34         tokenText.append(ch);
35     }
36     break;
```

接着，我们运行示例代码，就会输出下面的信息：

 复制代码

```
1 Int          int
2 Identifier    age
3 Assignment    =
4 IntLiteral    45
```

而当你试着解析 “intA = 10” 程序的时候，会把 intA 解析成一个标识符。输出如下：

 复制代码

```
1 Identifier    intA
2 Assignment    =
3 IntLiteral    10
```

解析算术表达式

解析完 “int age = 40” 之后，我们再按照上面的方法增加一些规则，这样就能处理算术表达式，例如 “2+3*5” 。增加的词法规则如下：

 复制代码

```
1 Plus : '+'
2 Minus : '-'
3 Star : '*'
4 Slash : '/'
```

然后再修改一下有限自动机和代码，就能解析 “2+3*5” 了，会得到下面的输出：

 复制代码

```
1 IntLiteral 2
```

```
2 Plus      +
3 IntLiteral 3
4 Star      *
5 IntLiteral 5
```

好了，现在我们已经能解析不少词法了，之后的课程里，我会带你实现一个公式计算器，所以在这里要先准备好所需要的词法分析功能。

课程小结

本节课，我们实现了一个简单的词法分析器。你可以看到，要实现一个词法分析器，首先需要写出每个词法的正则表达式，并画出有限自动机，之后，只要用代码表示这种状态迁移过程就可以了。

我们总是说理解原理以后，实现并不困难。今天的分享，你一定有所共鸣。

反之，如果你在编程工作中遇到困难，往往是因为不清楚原理，没有将原理吃透。而这门课就是要帮助你真正吃透编译技术中的几个核心原理，让你将知识应用到实际工作中，解决工作中遇到的困难。

小试了词法分析器之后，在下一讲，我会带你手工打造一下语法分析器，并实现一个公式计算器的功能。

一课一思

很多同学已经用到过正则表达式，这是学计算机必懂的知识点，十分有用。正则表达式工具其实就可以看做一个通用的词法分析器。那么你都用正则表达式功能做过哪些事情？有没有发现一些软件工具因为支持正则表达式而变得特别强大的情况呢？可以在留言区与大家一起交流。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

另外，为了便于你更好地学习，我将本节课的示例程序放到了 [🔗 GitHub](#) 上，你可以看一下。

精选留言 (101)



KnowNothing

2019-08-16

老师，在关键字和保留字的识别上，我认为有不需加入中间状态的更简单的方式：
完成词法分析后，遍历所有ID token，如果其text在关键字和保留字集合内，将该token类型改成对应的关键字/保留字类型。

或者，

每当识别出一个ID token，都检查其text，如果是在关键字和保留字集合内，纠正type。

作者回复：没错，可以的。

但是构造有限自动机的话，程序就可以标准化处理。不需要再手写其他代码。比如正则表达式工具。

当然，如果纯手写词法分析器，不受任何标准算法的限制的话，那发挥空间就会大很多。

爱动脑的好同学！

共 13 条评论 >

👍 58



傲娇的小宝

2019-08-18

感觉有限状态机有点类似图灵机的工作方式。我一般只用正则匹配一下文件名或者某个字符串是否符合我的预期。

作者回复：有限自动机是比较简单的一种自动机，对应于正则文法，也叫做3型文法。

比它强大的是下推自动机，对应于上下文无关文法，也叫做2型文法。

比它更强大的是线性有界自动机，对应于上下文相关文法，也叫1型文法。

图灵机的范围比它们都大，它对应0型文法。你任何能用产生式写出来的文法规则，都属于0型文法。

希望对你有帮助，了解有限自动机和图灵机的关系：)

共 2 条评论 >

👍 56



易林林

2019-08-17

宫老师，例子里面的词法分析大多是靠条件判断来实现，如果对一门完整的语言来进行分析的话，工作量会不会很大。我在想，是否有其他方式可以实现？

作者回复：课程的示例代码的主要目的是把意思讲明白，我甚至把状态都用枚举表示，就是为了易读。性能不是第一考虑。

从性能的角度，词法分析可以用查表的方法实现状态迁移。在每个状态，接收什么字符，切换到另外的状态。那样更快，这是常用的方法。

不光词法分析可以这么做，语法分析也可以。基于表驱动。这时候，最重要的是构造那张表。代码的话，就不大看明白是啥意思。



👍 48



Fan

2019-08-16

宫老师，有没有一些词法分析的demo可以推荐看看呢？

作者回复：最好的demo，就是现有语言的词法分析器，比如Java的、GNU c的，都能拿到源代码。比如Java的编译器在JDK的源代码里就有。

此外，我们在后端时会讲到LLVM工具。LLVM的文档里有一个小的教程，做了一个完整的前端。你也可以参考一下。<http://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html>

回头我整理一份清单放到github上，告诉大家去哪里下载。你的需求估计其他同学也有。

谢谢你！

共 2 条评论 >

👍 31



逗逼师父

2019-08-17

老师您好，我的疑问是，age>=45的有限状态机图中，为什么比较操作符不像标识符那样停留在同一个状态？我觉得>和>=都是属于比较操作符呀

作者回复：很好的问题。

是这样的。从Token分类的角度，我们确实可以把这两个归为一类。

但如果把它们看做同一个状态，就会有一些问题。比如，接收到=号应该怎么处理呢？接收第一个=号，仍然处于比较操作符状态。那么接收第二个呢？问题是，有限状态机接收字符的时候，是没法

数个数的。如果你要记个数，也就相当于在内部新增加了一个状态，还是等价于两个状态。我这么说你理解吗？

共 5 条评论 >

👍 22



诺亚

2019-08-25

isAlpha 的 alpha 好像没有字母的意思吧？用 alphabet 会不会比较合适点？

作者回复：你很细心，所以我也仔细给你解答下:-)

isAlpha是 is alphabetic 的缩写。isalpha()函数是好几个语言的标准库里都提供的，比如C、python 等。

alphabet指的是整个字母表，不是字母表里的单个字母。

共 2 条评论 >

👍 19



宋健

2020-03-24

老师，我写完这一节激动的浑身发抖，自己果然实现了一个简单的词法分析器！老师讲得太棒了！

作者回复：主要是你自己的功劳:-)

在技术领域，有时候你会觉得某个领域高山仰止，其实你自己也可以成为高山上的一棵青松。知识这东西，就在那里，只要想学，没有可能学不会。一旦学会，没有可能再变得不会，是个只会增加的过程，这是多便宜的事情！

不过，学习过程中，肯定还是会遇到挫折的，会觉得难懂，会觉得坚持不下去。这也没关系。你吃的苦越多，进入的境界就越高，这都是值得的！



👍 18



梓航(____)

2019-08-17

我是来提意见的，麻烦老师在讲示例的时候，把对应的github链接贴上，而不是在最后贴一个总的地址，我点进去一脸懵，哪个文件对应哪个例子啊？

作者回复：好的，感谢您提意见！已通知后台调整一下。

02课的文件是目录中的SimpleLexer.java文件。

另，如果到github的<https://github.com/RichardGong/PlayWithCompiler>项目主页，里面有每堂课的课件的介绍，供您参考。



15



请叫我康哥

2019-08-28

您好，老师，初步接触编译原理，您讲的手工打造词法分析，是用已知的java或c或其他语言实现，我想问一下，c语言的编译器是用什么实现的呢，或者其他语言的编译器？另外其他的词法分析器又是用什么语言编写的呢，谢谢

作者回复：在编译领域，有一个事情，叫做自举（bootstrapping），也就是这门语言的编译器可以用自己这门语言编写。这是语言迈向成熟的标志。一般前面的版本，是要借助别的语言编写编译器，但后面就应该用自己的语言来编译了。

著名的语言都实现了自举。比如，go语言的编译器是用go编写的（早期版本应该就是用C语言写的编译器。能实现自举，还是go发展历程上的一个里程碑），jdk里面自带了java语言的编译器，本身也是用java写的。

更早的语言，那当然是用汇编写编译器。比尔盖茨当年就是用汇编写。

掌握编译原理之后，其实用什么语言都可以写。

这门课程的示例语言是playscript。我计划后面把playscript实现自举。

共 2 条评论 >

15



(´□`) へー...

2019-10-03

有回随手发现Google搜索可以使用正则表达式.....然后感觉星星都亮了

作者回复：嗯。等你学了算法篇第16讲，了解了正则表达式的机制后，可以设计点正则表达式测试一下谷歌的性能，看看能否把谷歌的服务器累趴下...



13



devna

2019-08-16

之前做的一个项目中有大量的历史遗留脚本是用Perl写的，于是主动去学了Perl，不得不说，Perl是我见过所有语言里对正则表达式支持最强的，效率也是最高的(我想可能是因为Perl在语言核心内置了正则表达式引擎，不像其他语言，是通过各种模块支持的)。

后来从Perl了解到《精通正则表达式》这本书，买来看了下，确实是很好的一本书。虽然读完很久很多东西都忘了，但是对正则表达式的理解深刻了很多。

作者回复: great!

基于你已有的积累，是否可以进一步想想，能否自己写一个支持正则的工具？比如像grep、sed这些超级命令一样。那几个命令被认为是神级的命令，就是因为支持正则表达式，让它们能适应各种情况。

共 2 条评论 >

👍 13



xindoo

2019-08-16

正则表达式匹配3的任意倍数 <https://www.zhihu.com/question/24824487>

作者回复: 哇，真好玩！

点赞！



👍 10



阿尔伯特

2019-08-31

Mark.

<https://github.com/albertabc/compiler/blob/master/main.cpp>

作者回复: 下载，编译，并运行了。

你提供的cmake配置很赞，让编译很顺畅！

也看出你C++功底很扎实呀！

再次点赞！

共 2 条评论 >

👍 8



斐波那契

2019-08-25

老师 这个有限自动机的状态确认是不是以空格为分界点？比如 `int age = 45` 在读i时进入int1状态 然后读n进入int2状态然后读t进入int3状态然后读到一个空格结束此时是在关键字状态 所以说int是一个关键字 然后在重新初始化状态开始读取age直到读到；为止？

作者回复: 不完全是。

1. int和age之间是以空格分割。更准确的说, 是以“即非字母又非数字的任何字符”作为分割点。因为只要是字母或数字, 就会一直停留在标识符的状态里。
 2. 等号前后可以不必有空格, 算法仍然能正确的切分开。
- 这样说, 能明白吗?



👍 8



哆啦B梦

2019-11-20

听了老师的讲解, 感觉没有想象的那么难, 不管好坏, 先敲一版出来总是没错的:

<https://github.com/zyfandtmx/compile/blob/master/src/main/java/mycompiler/lexical/LexicalAnalysis.java>

作者回复: 看到了, 不错!

破山中贼易, 破心中贼难。

任何高峰, 都是能找到一条缓坡上去的。



👍 4



sheeeep

2019-08-16

ts的实现, 请老师和同学指正。 <https://github.com/sheeeep/fundamentals-of-compiling/blob/Ch02/src/lexical-analyser.ts>

作者回复: 非常好! 要号召大家跟你学习!

太棒了!

你有精力的话, 还可以再精进一下, 参考一下成熟编译器的词法分析工具, 从课程示例的代码的基础上再提升一个等级:)

比如说, 另一个同学提到过, 如何提升词法分析的性能什么的。

共 4 条评论 >

👍 4



冬瓜

2019-08-16

int 后面的 id 的正则是不是 `[a-zA-Z_][0-9a-zA-Z_]*` 这样就行，为啥要将数字通过 `|` 连接呢？是不是有什么用意 😏

作者回复：就是个习惯而已，把数字和字符分两组。

我们在写正式的词法文件时，有时会这么写：

Id: Charactor (Charactor | Digit)*

Number: Digit+

Charactor: [a-zA-Z_]

Digit: [0-9]

这时候，Digit 在几个不同的规则中是复用的。

共 4 条评论 >

👍 4



鱼_XueTr

2019-08-16

正则表达式在做爬虫和文本处理中用过比较多。

作者回复：是滴。

程序编译的第一阶段工作，本质也是文本处理。



👍 4



许童童

2019-08-16

正则表达式在实现中一般有两种：NFA 和 DFA，一般语言用 DFA，但会有回溯的性能问题，用的时候一定要注意。

作者回复：如果要实现一个通用的正则表达式工具，这时候没办法手工构造 DFA，会遇到你说的情况。

如果往深看一步，词法分析和我们后面讲的自顶向下的语法分析有共同之处，所以都会有回溯的可能性。

一般语言的词法，都不会太复杂，并且可以手工构造 DFA，所以也就可以尽量避免回溯了。



👍 3



恩佐

2019-11-01

https://github.com/shaojintian/learn_compiler

老师能看看我的lexer么？用golang写的

请老师和同学们指正，感谢

作者回复: 看了你的代码，挺不错的。如果再补充一下README，让人可以迅速build和测试你的项目就更好了。

共 2 条评论 >

👍 2