

07 | 编译器前端工具（二）：用Antlr重构脚本语言

2019-08-28 宫文学 来自北京

《编译原理之美》



上一讲，我带你用 Antlr 生成了词法分析器和语法分析器，也带你分析了，跟一门成熟的语言相比，在词法规则和语法规则方面要做的一些工作。

在词法方面，我们参考 Java 的词法规则文件，形成了一个 CommonLexer.g4 词法文件。在这个过程中，我们研究了更完善的字符串字面量的词法规则，还讲到要通过规则声明的前后顺序来解决优先级问题，比如关键字的规则一定要在标识符的前面。

目前来讲，我们已经完善了词法规则，所以今天我们来补充和完善一下语法规则，看一看怎样用最高效的速度，完善语法功能。比如一天之内，我们是否能为某个需要编译技术的项目实现一个可行性原型？

而且，我还会带你熟悉一下常见语法设计的最佳实践。这样当后面的项目需要编译技术做支撑时，你就会很快上手，做出成绩了！

接下来，我们先把表达式的语法规则梳理一遍，让它达到成熟语言的级别，然后再把语句梳理一遍，包括前面几乎没有讲过的流程控制语句。最后再升级解释器，用 Visitor 模式实现对 AST 的访问，这样我们的代码会更清晰，更容易维护了。


好了，让我们正式进入课程，先将表达式的语法完善一下吧！

完善表达式 (Expression) 的语法

在“[06 | 编译器前端工具（一）：用 Antlr 生成词法、语法分析器](#)”中，我提到 Antlr 能自动处理左递归的问题，所以在写表达式时，我们可以大胆地写成左递归的形式，节省时间。

但这样，我们还是要为每个运算写一个规则，逻辑运算写完了要写加法运算，加法运算写完了写乘法运算，这样才能实现对优先级的支持，还是有些麻烦。

其实，Antlr 能进一步地帮助我们。我们可以把所有的运算都用一个语法规则来涵盖，然后用最简洁的方式支持表达式的优先级和结合性。在我建立的 PlayScript.g4 语法规则文件中，只用了一小段代码就将所有的表达式规则描述完了：

 复制代码

```
1 expression
2     : primary
3     | expression bop='.'
4     | ( IDENTIFIER
5       | functionCall
6       | THIS
7       )
8     | expression '[' expression ']'
9     | functionCall
10    | expression postfix=('++' | '--')
11    | prefix=('+' | '-' | '++' | '--') expression
12    | prefix=('~' | '!') expression
13    | expression bop=('*' | '/' | '%') expression
14    | expression bop=('+' | '-') expression
15    | expression ('<' '<' | '>' '>' '>' | '>' '>') expression
16    | expression bop('<=' | '>=' | '>' | '<') expression
17    | expression bop=INSTANCEOF typeType
18    | expression bop=('==' | '!=') expression
19    | expression bop='&' expression
20    | expression bop='^' expression
21    | expression bop='|' expression
```

```


22 | expression bop='&&' expression
23 | expression bop='||' expression
24 | expression bop='?' expression ':' expression
25 | <assoc=right> expression
26 | bop=('=' | '+=' | '-=' | '*=' | '/=' | '&=' | '|=' | '^=' | '>>=' | '>>>='
27 | expression
28 ;

```

这个文件几乎包括了我们需要的所有表达式规则，包括几乎没提到的点符号表达式、递增和递减表达式、数组表达式、位运算表达式规则等，已经很完善了。

那么它是怎样支持优先级的呢？原来，优先级是通过右侧不同产生式的顺序决定的。在标准的上下文无关文法中，产生式的顺序是无关的，但在具体的算法中，会按照确定的顺序来尝试各个产生式。

你不可能一会儿按这个顺序，一会儿按那个顺序。然而，同样的文法，按照不同的顺序来推导的时候，得到的 AST 可能是不同的。我们需要注意，这一点从文法理论的角度，是无法接受的，但从实践的角度，是可以接受的。比如 LL 文法和 LR 文法的概念，是指这个文法在 LL 算法或 LR 算法下是工作正常的。又比如我们之前做加法运算的那个文法，就是递归项放在右边的那个，在递归下降算法中会引起结合性的错误，但是如果用 LR 算法，就完全没有这个问题，生成的 AST 完全正确。

 复制代码

```

1 additiveExpression
2   :   IntLiteral
3   |   IntLiteral Plus additiveExpression
4   ;

```

Antlr 的这个语法实际上是把产生式的顺序赋予了额外的含义，用来表示优先级，提供给算法。所以，我们可以说这些文法是 Antlr 文法，因为是与 Antlr 的算法相匹配的。当然，这只是我起的一个名字，方便你理解，免得你产生困扰。

我们再来看看 Antlr 是如何依据这个语法规则实现结合性的。在语法文件中，Antlr 对于赋值表达式做了 <assoc=right> 的属性标注，说明赋值表达式是右结合的。如果不标注，就是左

结合的，交给 Antlr 实现了！

我们不妨继续猜测一下 Antlr 内部的实现机制。我们已经分析了保证正确的结合性的算法，比如把递归转化成循环，然后在构造 AST 时，确定正确的父子节点关系。那么 Antlr 是不是也采用了这样的思路呢？或者说还有其它方法？你可以去看看 Antlr 生成的代码验证一下。

在思考这个问题的同时你会发现，**学习原理是很有用的**。因为当你面对 Antlr 这样工具时，能够猜出它的实现机制。


通过这个简化的算法，AST 被成功简化，不再有加法节点、乘法节点等各种不同的节点，而是统一为表达式节点。你可能会问了：“如果都是同样的表达式节点，怎么在解析器里把它们区分开呢？怎么知道哪个节点是做加法运算或乘法运算呢？”

很简单，我们可以查找一下当前节点有没有某个运算符的 Token。比如，如果出现了或者运算的 Token（“||”），就是做逻辑或运算，而且语法里面的 bop=、postfix=、prefix= 这些属性，作为某些运算符 Token 的别名，也会成为表达式节点的属性。通过查询这些属性的值，你可以很快确定当前运算的类型。

到目前为止，我们彻底完成了表达式的语法工作，可以放心大胆地在脚本语言里使用各种表达式，把精力放在完善各类语句的语法工作上了。

完善各类语句 (Statement) 的语法

我先带你分析一下 PlayScript.g4 文件中语句的规则：

 复制代码

```
1 statement
2     : blockLabel=block
3     | IF parExpression statement (ELSE statement)?
4     | FOR '(' forControl ')' statement
5     | WHILE parExpression statement
6     | DO statement WHILE parExpression ';'
7     | SWITCH parExpression '{' switchBlockStatementGroup* switchLabel* '}'
8     | RETURN expression? ';'
9     | BREAK IDENTIFIER? ';'
10    | SEMI
```

```
11      | statementExpression=expression ';'
12      ;
```

同表达式一样，一个 statement 规则就可以涵盖各类常用语句，包括 if 语句、for 循环语句、while 循环语句、switch 语句、return 语句等等。表达式后面加一个分号，也是一种语句，叫做表达式语句。


从语法分析的难度来看，上面这些语句的语法比表达式的语法简单的多，左递归、优先级和结合性的问题这里都没有出现。这也算先难后易，苦尽甘来了吧。实际上，我们后面要设计的很多语法，都没有想象中那么复杂。

既然我们尝到了一些甜头，不如趁热打铁，深入研究一下 if 语句和 for 语句？看看怎么写这些语句的规则？多做这样的训练，再看到这些语句，你的脑海里就能马上反映出它的语法规则。

1. 研究一下 if 语句


在 C 和 Java 等语言中，if 语句通常写成下面的样子：

```
1  if (condition)
2      做一件事情；
3  else
4      做另一件事情；
```


 复制代码

但更多情况下，if 和 else 后面是花括号起止的一个语句块，比如：

```
1  if (condition){
2      做一些事情；
3  }
4  else{
5      做另一些事情；
6  }
```

 复制代码

它的语法规则是这样的：

 复制代码


```
1 statement :  
2     ...  
3     | IF parExpression statement (ELSE statement)?  
4     ...  
5     ;  
6 parExpression : '(' expression ')';
```

我们用了 IF 和 ELSE 这两个关键字，也复用了已经定义好的语句规则和表达式规则。你看，语句规则和表达式规则一旦设计完毕，就可以被其他语法规则复用，多么省心！

但是 if 语句也有让人不省心的地方，比如会涉及到二义性文法问题。所以，接下来我们就借 if 语句，分析一下二义性文法这个现象。

2. 解决二义性文法

学计算机语言的时候，提到 if 语句，会特别提一下嵌套 if 语句和悬挂 else 的情况，比如下面这段代码：

 复制代码

```
1 if (a > b)  
2 if (c > d)  
3 做一些事情;  
4 else  
5 做另外一些事情;
```

在上面的代码中，我故意取消了代码的缩进。那么，你能不能看出 else 是跟哪个 if 配对的呢？

一旦你语法规则写得不够好，就很可能形成二义性，也就是用同一个语法规则可以推导出两个不同的句子，或者说生成两个不同的 AST。这种文法叫做二义性文法，比如下面这种写法：

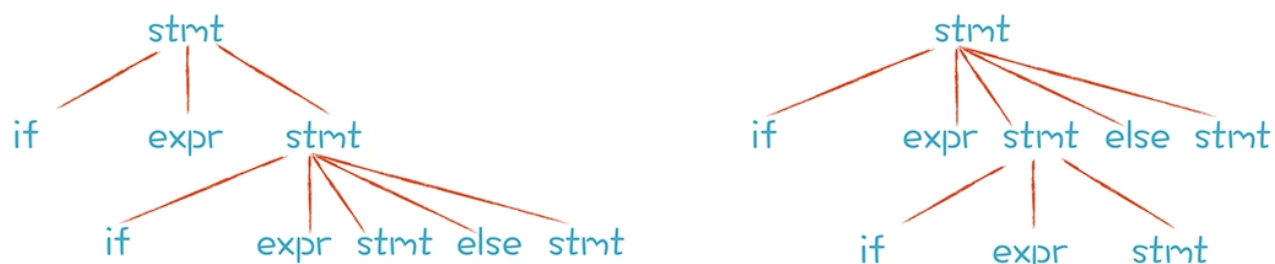

```

1 stmt -> if expr stmt
2       | if expr stmt else stmt
3       | other

```

[复制代码](#)

按照这个语法规则，先采用第一条产生式推导或先采用第二条产生式推导，会得到不同的 AST。左边的这棵 AST 中，else 跟第二个 if 配对；右边的这棵 AST 中，else 跟第一个 if 配对。



大多数高级语言在解析这个示例代码时都会产生第一个 AST，即 else 跟最邻近的 if 配对，也就是下面这段带缩进的代码表达的意思：

```

1 if (a > b)
2   if (c > d)
3     做一些事情;
4   else
5     做另外一些事情;

```

[复制代码](#)

那么，有没有办法把语法写成没有二义性的呢？当然有了。

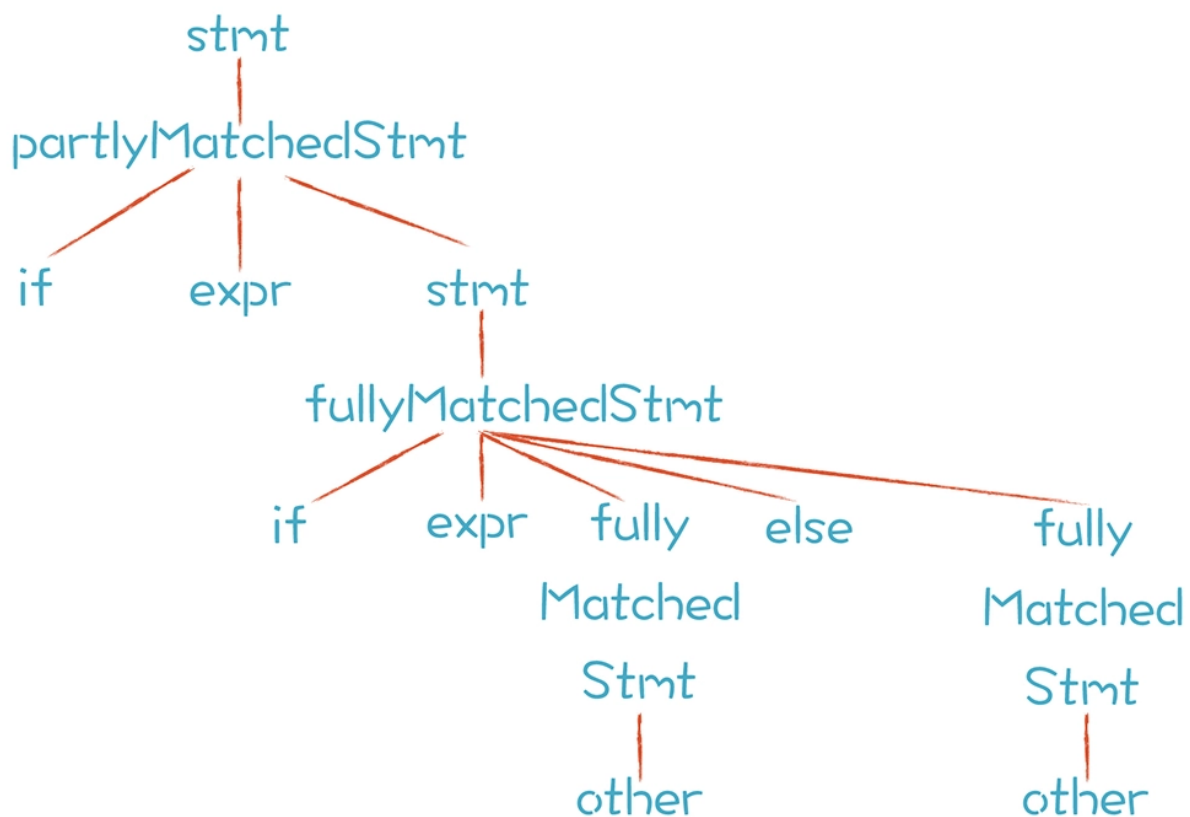
```

1 stmt -> fullyMatchedStmt | partlyMatchedStmt
2 fullyMatchedStmt -> if expr fullyMatchedStmt else fullyMatchedStmt
3                   | other
4 partlyMatchedStmt -> if expr stmt
5                   | if expr fullyMatchedStmt else partlyMatchedStmt

```

[复制代码](#)

按照上面的语法规则，只有唯一的推导方式，也只能生成唯一的 AST：



其中，解析第一个 if 语句时只能应用 partlyMatchedStmt 规则，解析第二个 if 语句时，只能适用 fullyMatchedStmt 规则。

这时，我们就知道可以通过改写语法规则来解决二义性文法。至于怎么改写规则，确实不像左递归那样有清晰的套路，但是可以多借鉴成熟的经验。

再说回我们给 Antlr 定义的语法，这个语法似乎并不复杂，怎么就能确保不出现二义性问题呢？因为 Antlr 解析语法时用到的是 LL 算法。

LL 算法是一个深度优先的算法，所以在解析到第一个 statement 时，就会建立下一级的 if 节点，在下一级节点里会把 else 子句解析掉。如果 Antlr 不用 LL 算法，就会产生二义性。这再次验证了我们前面说的那个知识点：文法要经常和解析算法配合。

分析完 if 语句，并借它说明了二义性文法之后，我们再针对 for 语句做一个案例研究。

3. 研究一下 for 语句


for 语句一般写成下面的样子：

```
1  for (int i = 0; i < 10; i++){
2      println(i);
3  }
```

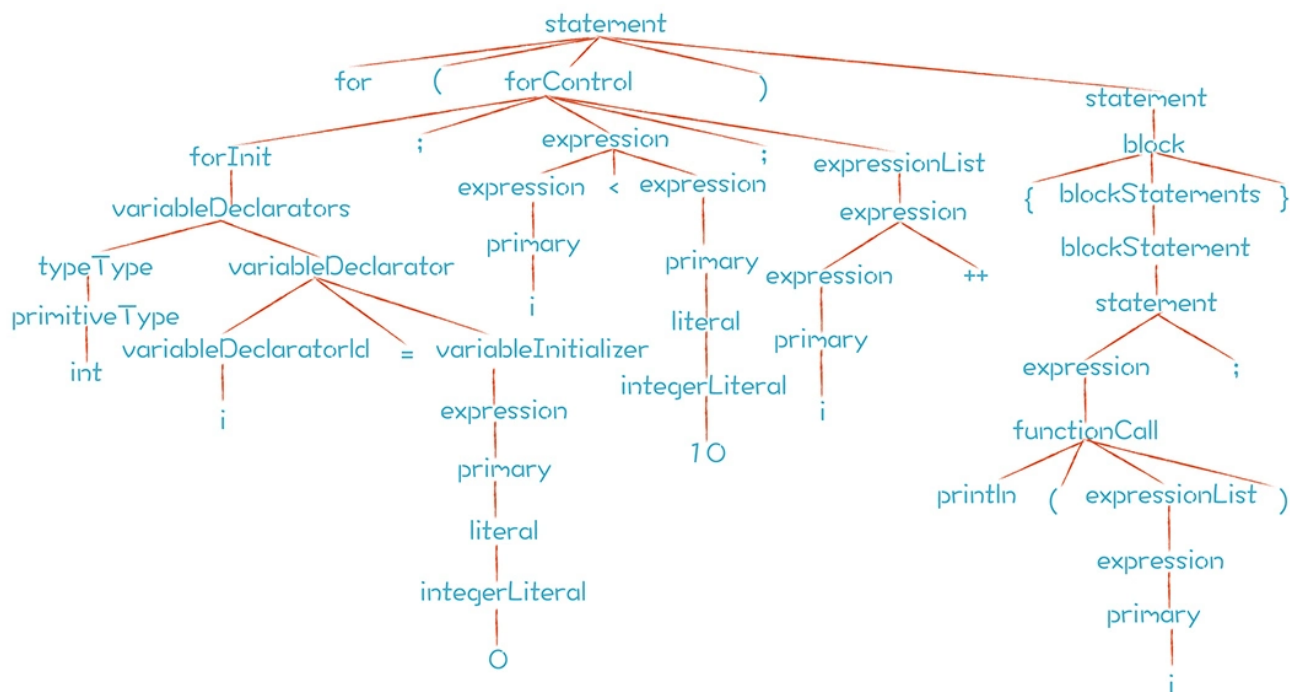
 复制代码

相关的语法规则如下：

```
1  statement :
2      ...
3      | FOR '(' forControl ')' statement
4      ...
5      ;
6
7  forControl
8      : forInit? ';' expression? ';' forUpdate=expressionList?
9      ;
10
11 forInit
12     : variableDeclarators
13     | expressionList
14     ;
15
16 expressionList
17     : expression (',' expression)*
18     ;
```

 复制代码

从上面的语法规则中看到，for 语句归根到底是由语句、表达式和变量声明构成的。代码中的 for 语句，解析后形成的 AST 如下：



熟悉了 for 语句的语法之后，我想提一下语句块 (block)。在 if 语句和 for 语句中，会用到它，所以我捎带着把语句块的语法构成写了一下，供你参考：

[复制代码](#)

```

1 block
2     : '{' blockStatements '}'
3     ;
4
5 blockStatements
6     : blockStatement*
7     ;
8
9 blockStatement
10    : variableDeclarators ';'      //变量声明
11    | statement
12    | functionDeclaration         //函数声明
13    | classDeclaration            //类声明
14    ;

```

现在，我们已经拥有了一个相当不错的语法体系，除了要放到后面去讲的函数、类有关的语法之外，我们几乎完成了 playscript 的所有的语法设计工作。接下来，我们再升级一下脚本解释器，让它能够支持更多的语法，同时通过使用 Visitor 模式，让代码结构更加完善。

用 Visitor 模式升级脚本解释器

我们在纯手工编写的脚本语言解释器里，用了一个 `evaluate()` 方法自上而下地遍历了整棵树。随着要处理的语法越来越多，这个方法的代码量会越来越大，不便于维护。而 Visitor 设计模式针对每一种 AST 节点，都会有一个单独的方法来负责处理，能够让代码更清晰，也更便于维护。

Antlr 能帮我们生成一个 Visitor 处理模式的框架，我们在命令行输入：

```
1 antlr -visitor PlayScript.g4
```

[📄 复制代码](#)

`-visitor` 参数告诉 Antlr 生成下面两个接口和类：

```
1 public interface PlayScriptVisitor<T> extends ParseTreeVisitor<T> {...}
2
3 public class PlayScriptBaseVisitor<T> extends AbstractParseTreeVisitor<T> impleme
```

[📄 复制代码](#)

在 `PlayScriptBaseVisitor` 中，可以看到很多 `visitXXX()` 这样的方法，每一种 AST 节点都对应一个方法，例如：

```
1 @Override public T visitPrimitiveType(PlayScriptParser.PrimitiveTypeContext ctx)
```

[📄 复制代码](#)


其中泛型 `< T >` 指的是访问每个节点时返回的数据的类型。在我们手工编写的版本里，当时只处理整数，所以返回值一律用 `Integer`，现在我们实现的版本要高级一点，AST 节点可能返回各种类型的数据，比如：

浮点型运算的时候，会返回浮点数；

字符类型运算的时候，会返回字符型数据；


还可能是程序员自己设计的类型，如某个类的实例。

所以，我们就让 Visitor 统一返回 Object 类型好了，能够适用于各种情况。这样，我们的 Visitor 就是下面的样子（泛型采用了 Object）：

 复制代码

```
1 public class MyVisitor extends PlayScriptBaseVisitor<Object>{
2     ...
3 }
```

这样，在 visitExpression() 方法中，我们可以编写各种表达式求值的代码，比如，加法和减法运算的代码如下：


 复制代码

```
1 public Object visitExpression(ExpressionContext ctx) {
2     Object rtn = null;
3     //二元表达式
4     if (ctx.bop != null && ctx.expression().size() >= 2) {
5         Object left = visitExpression(ctx.expression(0));
6         Object right = visitExpression(ctx.expression(1));
7         ...
8         Type type = cr.node2Type.get(ctx); //数据类型是语义分析的成果
9
10        switch (ctx.bop.getType()) {
11            case PlayScriptParser.ADD: //加法运算
12                rtn = add(leftObject, rightObject, type);
13                break;
14            case PlayScriptParser.SUB: //减法运算
15                rtn = minus(leftObject, rightObject, type);
16                break;
17            ...
18        }
19    }
20    ...
21 }
```

其中 ExpressionContext 就是 AST 中表达式的节点，叫做 Context，意思是你能从中取出这个节点所有的上下文信息，包括父节点、子节点等。其中，每个子节点的名称跟语法中的名称


是一致的，比如加减法语法规则是下面这样：

```
1 expression bop=('+' | '-' ) expression
```

 复制代码

那么我们可以用 ExpressionContext 的这些方法访问子节点：


```
1 ctx.expression();           //返回一个列表，里面有两个成员，分别是左右两边的子节点
2 ctx.expression(0);          //运算符左边的表达式，是另一个ExpressionContext对象
3 ctx.expression(1);          //云算法右边的表达式
4 ctx.bop();                   //一个Token对象，其类型是PlayScriptParser.ADD或SUB
5 ctx.ADD();                   //访问ADD终结符，当做加法运算的时候，该方法返回非空值
6 ctx.MINUS();                 //访问MINUS终结符
```

 复制代码

在做加法运算的时候我们还可以递归的对下级节点求值，就像代码里的 visitExpression(ctx.expression(0))。同样，要想运行整个脚本，我们只需要 visit 根节点就行了。

所以，我们可以用这样的方式，为每个 AST 节点实现一个 visit 方法。从而把整个解释器升级一遍。除了实现表达式求值，我们还可以为今天设计的 if 语句、for 语句来编写求值逻辑。以 for 语句为例，代码如下：

```
1 // 初始化部分执行一次
2 if (forControl.forInit() != null) {
3     rtn = visitForInit(forControl.forInit());
4 }
5
6 while (true) {
7     Boolean condition = true; // 如果没有条件判断部分，意味着一直循环
8     if (forControl.expression() != null) {
9         condition = (Boolean) visitExpression(forControl.expression());
10    }
11
12    if (condition) {
13        // 执行for的语句体
```

 复制代码

```
14         rtn = visitStatement(ctx.statement(0));
15
16         // 执行forUpdate, 通常是“i++”这样的语句。这个执行顺序不能出错。
17         if (forControl.forUpdate != null) {
18             visitExpressionList(forControl.forUpdate);
19         }
20     } else {
21         break;
22     }
23 }
```

你需要注意 for 语句中各个部分的执行规则，比如：

forInit 部分只能执行一次；

每次循环都要执行一次 forControl，看看是否继续循环；

接着执行 for 语句中的语句体；

最后执行 forUpdate 部分，通常是一些 “i++” 这样的语句。

支持了这些流程控制语句以后，我们的脚本语言就更丰富了！

课程小结

今天，我带你用 Antlr 高效地完成了很多语法分析工作，比如完善表达式体系，完善语句体系。除此之外，我们还升级了脚本解释器，使它能够执行更多的表达式和语句。

在实际工作中，针对面临的具体问题，我们完全可以像今天这样迅速地建立可以运行的代码，专注于解决领域问题，快速发挥编译技术的威力。

而且在使用工具时，针对工具的某个特性，比如对优先级和结合性的支持，我们大致能够猜到工具内部的实现机制，因为我们已经了解了相关原理。

一课一思

我们通过 Antlr 并借鉴成熟的规则文件，很快就重构了脚本解释器，这样工作效率很高。那么，针对要解决的领域问题，你是不是借鉴过一些成熟实践或者最佳实践来提升效率和质量？

在这个过程中又有什么心得呢？欢迎在留言区分享你的心得。

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。

我把一门功能比较全的脚本语言的示例放在了 playscript-java 项目下，以后几讲的内容都会参考这里面的示例代码。

playscript-java（项目目录）：[码云](#) [GitHub](#)

PlayScript.java（入口程序）：[码云](#) [GitHub](#)

PlayScript.g4（语法规则）：[码云](#) [GitHub](#)

ASTEvaluator.java（解释器）：[码云](#) [GitHub](#)

© 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。

精选留言 (35)



李懂

2019-08-29

现在都是用一门语言去实现这些功能，我想知道最开始的语言是怎么实现分析的呢！有一点鸡生蛋蛋生鸡！

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）。

最早的语言的编译器，那肯定是用汇编写。到一定程度后再自举。

共 2 条评论 >

👍 20



Spring

2019-08-28

老师，你好。请教一下，词法，语法解析后生成 AST 后，计算机怎么指导我的AST 中的“+”就是执行 add 的计算呢？这其中是不是还有还存在一个中间层？

作者回复：对。你提的问题很好。

说明你思考的很深入了。

“+”执行加法运算，是由计算机语言的语义规定的。比如，你可以再让“+”去做字符串连接，这也是语义上的规定。

所以，计算机语言之间真正的差别，其实在语义上。

词法分析、语法分析完毕以后，只是搭起一个数据结构。至于基于这个结构可以干什么，还必须附加语义。你可以在这个AST上附加一些“动作指令”，比如对AST遍历的时候，遍历到“+”，就把两边加起来。这就是属性计算做的事情。我们把value作为一个属性，用一些规则来计算属性。说起来，属性计算还是大师高德纳提出来的。

你再沿着自己的思路深入下去，你可能自己把高德纳大师想到的也都想出来了。

看来你对编译原理的直觉很好:-D



17



Void_seT

2019-08-28

老师，目前的学习过程中，比如表达式语法规则、语句语法规则等，虽然能知道它们表示了什么，但是并不知道它是怎么凭空产生的；请问：这种规则是相对比较固定的，我们要使用时，可以参照“标准”的规则文法进行修改呢？还是要自己掌握各种类型语法规则的各个组成细节，以便于在写语法规则时可以信手拈来呢？如果需要熟练掌握语法规则的各个组成细节，目前的工作如果还用不到生成“小编译器”这种技能，也就是没有练习或高强度的训练时间的话，是否需要现在就硬啃下这块硬骨头（因为怕长时间不使用，将来真正要使用时，还是要重新再训练一遍）？

作者回复：各种文法规则的设计经验的积累，属于“最佳实践”的范畴。我建议大家不仅仅是要懂原理，还能掌握一些最佳实践，说起某个语法现象的时候，随后就能写出几个文法来。

能有这种实操能力，才算是把理论落到实际了。这些“最佳实践”，属于你自己积累的领域经验，这也是你为什么会有竞争力的原因。

这些经验，只有动手，多看别人的，才能积累。一般没有书籍专门讲这个，顶多是以示例的方式呈现。



15



宇智波芭芭干

2019-09-10

学习时总感觉节奏在老师那边，自己的思路并不连贯，对于初学者容易出现断片。在极客时间其它老师那里也同步购买了linux以及网络协议，另外一边通过故事的形式通熟易懂的讲解了一些底层知识原理，学习也是相当顺畅有兴趣，而这里不知道为啥就是顺畅不起来，差距不是一般的大。

作者回复: 谢谢提意见。我们会收集大家的意见，在课件版本迭代时提升表述水平！

共 7 条评论 >



windpiaoxue

2019-08-31

老师您好

例如下面这个规则：

```
stmt -> if expr stmt
      | if expr stmt else stmt
      | other
```

我测试了一下，antlr使用上面这个规则可以正确的处理悬挂else的问题，antlr在处理这种二义性问题的时候，是依据什么来处理的。

作者回复: 为你的动手实践点赞！

其实原因我在文稿里已经说了。

我们实现一个算法的时候，是有确定的顺序来匹配的。所以，即使是二义性文法，在某种算法下也可以正常解析。

严格的非二义性文法要求得比较高。它要求是算法无关的。也就是不管你用最左还是最右推导，得出的结果是一样的。

关键点，在于把“文法”和“算法”这两件事区分开。文法是二义的，用某个具体算法却不一定是二义的。

其余的部分，你可以再看看文稿，是否能理解。Antlr是LL算法，最左推导、深度优先。如果你一时看不明白，也没关系，因为到后面我还会专门讲LL算法。



Geek_6304e3

2022-02-11

老师，后面的用 Vistor 模式升级脚本解释器开始有点看不懂，不懂java，有JavaScript版本的

吗？文章一些代码都是直接说这样写，但是我不知道这样写之后在哪里关联运行起来。



4



Geek_6304e3

2022-02-11

visitor这些解释器要怎么执行呢？



2



Geek_6304e3

2022-02-11

visitExpression方法实在哪个文件生成的？文中没有说。



2



fung

2019-11-05

老师，这一段看不懂咋办，有的救吗？看不懂这些语法啊，能解析下吗？或有其他资料介绍吗？谢谢

```
expression : primary | expression bop='.' ( IDENTIFIER | functionCall | THIS ) | expression  
            '[' expression ']' | functionCall | expression postfix=('++' | '--') | prefix=('+' | '-' | ++ | --) exp  
ression | prefix=('~' | '!') expression | expression bop=('*' | '/' | '%') expression | expression bo  
p=('+' | '-') expression | expression ('<' | '<' | '>' | '>' | '>' | '>' | '>' | '>') expression | expression bop=  
('<=' | '>=' | '>' | '<') expression .....
```

作者回复: 首先，关于Antlr的详细语法，你可以看一下它的作者的一本书：《the definitive antlr 4 reference》，应该也有中文版的。

另外，你可以搜一下EBNF的语法，因为antlr的语法基本上就是EBNF的语法，跟正则表达式的语法也很像，然后又加了一些元素，比如给某些部分做了命名。

bop=('+' | '-')是给('+' | '-')起了个名称，便于引用。

最后，当你动手实践的时候，这些困难就都不存在了。你就是对它们陌生。用多了就不陌生了！

共 2 条评论 >

2



shantelle

2019-10-25

宫老师你好，请问这个匹配的是什么内容呢

'\ ' [btnfr""\]

作者回复: 转义字符, 比如: \t是tab。



👍 2



zhj

2019-10-16

现在拿到的ASTEvaluator, 都裹扎了编译器相关的代码, 这里才看到Ast树, 这边没有很好的版本迭代吗, 上来直接就是讲课同步的代码, 看的云里雾里, 没法循序渐进

作者回复: 回头把代码分拆整理一下。



👍 2



石维康

2019-08-28

statement

```
: blockLabel=block
| IF parExpression statement (ELSE statement)?
| FOR '(' forControl ')' statement
| WHILE parExpression statement
| DO statement WHILE parExpression ';'
| SWITCH parExpression '{' switchBlockStatementGroup* switchLabel* '}'
| RETURN expression? ';'
| BREAK IDENTIFIER? ';'
| SEMI
| statementExpression=expression ';'
;
```

请问" : blockLabel=block"这个规则如何解释?谢谢!

作者回复: 这是给block起了个别名, 这样在生成的AST节点StatementContext中, 就会有blockLabel这个属性, 来访问这个下级节点。

就是为了编程方便的。



👍 2



2022-02-11

老师，你给的java版本的项目跑不起来。。不懂java



1



草戊

2019-12-15

老师，有些语言的部分文法是上下文有关，比如说必须是第七列写*号来注释。对于这样的语言分析，有什么好的建议吗？在parser前先做预处理变换？

作者回复：如果是手写编译器，就很容易处理。在处理*号的时候加一点代码进行上下文的分析就好了。

像Antlr这样编译器生成工具，支持你在做解析的时候嵌入自己的代码，进行与上下文有关的分析。分析的结果，会反馈回来影响编译过程。

所以，在词法或语法分析时就开始进行上下文的分析（或语义分析），是一个普遍使用的技巧。

如果你想深入了解一下这个问题，推荐你看一下这篇论文：

<https://wwwantlr.org/papers/predicated-parsing.pdf>

这里面还有其他一些例子。



1



cry soul

2019-10-05

建议老师用git搭好tag来表示每个课程到到哪部分源码，不然需要读好几篇才能自己尝试。

作者回复：谢谢你的建议。有的代码文件确实很长，查找不太容易。我后面优化一下代码链接！



1



李懂

2019-08-29

JavaScript中的this是咋实现的，这个一直处于迷糊当中，好想弄清楚，不同语言之间语意的差别，学完语意能理解么😂😂😂，看了很多课程，都很失望，都是再讲几种场景，怎么指向，没实质的改变！

作者回复：我记着你这个需求。

我看看能否把这个点插到某一讲中。

共 4 条评论 >



1



中年男子

2019-08-28

编译 git 里 PlayScript-cpp, 我这里报错, PlayScriptJit.h 这个文件, 搞了半天没搞懂
In file included from /Users/shiny/learn/PlayWithCompiler/playscript-cpp/src/PlayScript.cpp:
5:

[build] In file included from /Users/shiny/learn/PlayWithCompiler/playscript-cpp/src/grammar/IRGen.h:28:

[build] /Users/shiny/learn/PlayWithCompiler/playscript-cpp/src/grammar/PlayScriptJIT.h:33:
31: error: unknown type name 'LegacyRTDyldObjectLinkingLayer'; did you mean 'RTDyldObjectLinkingLayer'?

[build] using ObjLayerT = LegacyRTDyldObjectLinkingLayer;

[build] ^~~~~~

[build] RTDyldObjectLinkingLayer

作者回复: 你的进度有点快!

playscript-cpp我还没有整理好。

如果你着急看后端的东西, 建议你先做两件事情:

- 1.用Antlr将.g4文件生成c++代码, 测试一下在C++中运行是否OK。
- 2.下载和安装LLVM, 做做教程里的例子, 有一个是c++的例子。

好消息是, 这两个项目都是用cmake管理的。



1



许童童

2019-08-28

难度越来越大了, 要好好消化才行。

作者回复: 我相信你的消化能力:-D



1



(_ _)

2019-08-28

写了一晚上终于用c语言模仿着实现了第二节课的内容

https://github.com/hongningexpro/Play_with_Compiler/tree/master/01-Simple_Lexer

作者回复: 点赞!

动手出真知!



1



Geek_3c4e22

2022-08-14 来自福建

压根就不知道怎么把代码跑起来

