让我们构建一个简单的解释器。第 16 部分：识别过程调用 (https://ruslanspivak.com/lsbasi-part16/)

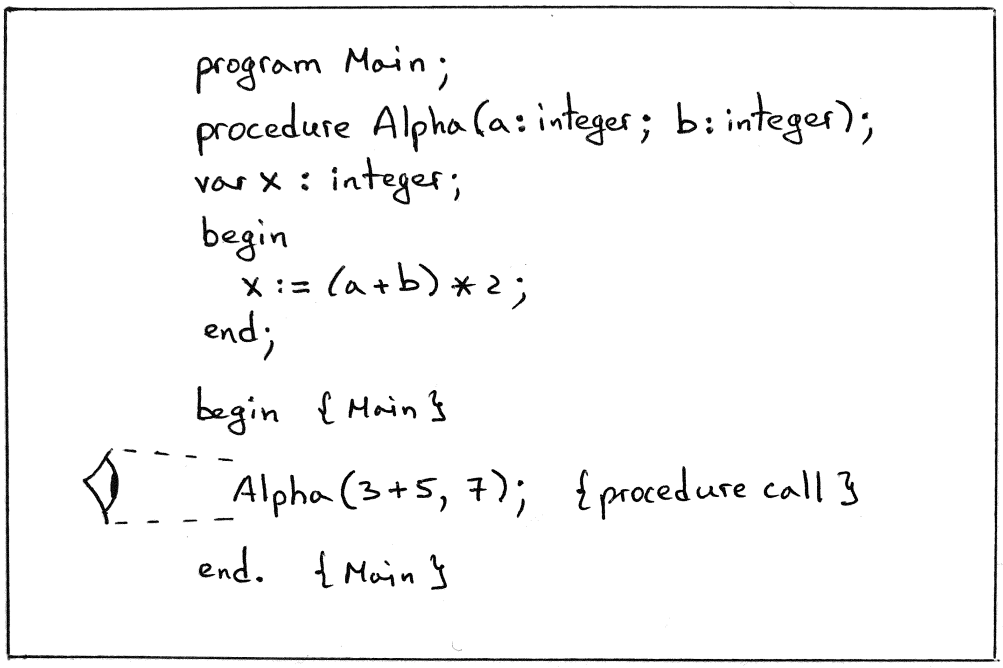
日期 2019 年 7 月 23 日，星期二

> “学习就像逆流而上：不进则退。” ——中国谚语

今天我们将扩展我们的解释器来识别过程调用。我希望你现在已经锻炼了你的编码肌肉并准备好解决这一步。这对我们来说是学习如何执行过程调用之前的必要步骤，这将是我们将在以后的文章中详细介绍的主题。

今天的目标是确保当我们的解释器读取带有过程调用的程序时，解析器构造一个抽象语法树（AST），并为过程调用构建一个新的树节点，而语义分析器和解释器不会走AST时抛出任何错误。

让我们看一个包含过程调用Alpha(3 + 5, 7)的示例程序：



让我们的口译员识别上述程序将是我们今天的重点。

与任何新功能一样，我们需要更新解释器的各种组件以支持此功能。让我们一一深入研究这些组件中的每一个。

首先，我们需要更新解析器。这是我们需要进行的所有解析器更改的列表，以便能够解析过程调用并构建正确的AST：

1. 我们需要添加一个新的AST节点来表示一个过程调用
2. 我们需要为过程调用语句添加新的语法规则；然后我们需要在代码中实现规则
3. 我们需要扩展语句语法规则以包含过程调用语句规则并更新语句方法以反映语法的变化

1. 让我们首先创建一个单独的类来表示过程调用AST节点。让我们调用类ProcedureCall：

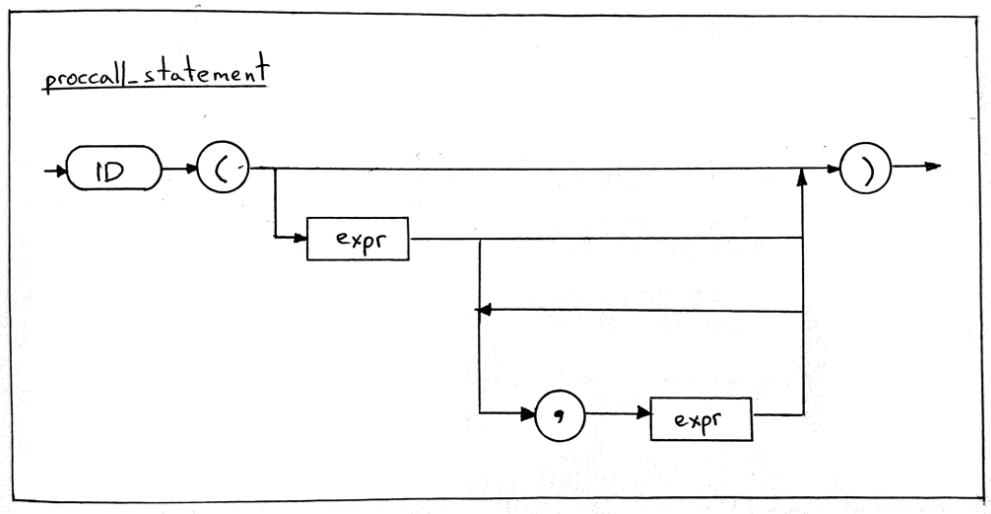
```
class ProcedureCall ( AST ):
    def __init__ ( self , proc_name , actual_params , token ):
        self 。proc_name = proc_name
        自我。actual_params = actual_params # AST 节点列表
        self . 令牌 = 令牌
```

该ProcedureCall类的构造函数有三个参数：程序名，实际参数列表（也称为参数），和令牌。这里没有什么特别的，只有足够的信息让我们捕获特定的过程调用。

2. 我们需要采取的下一步是扩展我们的语法并为过程调用添加语法规则。让我们调用规则proccall_statement：

```
proccall_statement : ID LPAREN ( expr ( COMMA expr )*)？ 帕伦
```

这是规则的相应语法图：



从上图可以看出，过程调用是一个ID标记，后跟一个左括号，后跟零个或多个用逗号分隔的表达式，再后跟一个右括号。以下是一些符合规则的过程调用示例：

```
阿尔法() ;
阿尔法( 1 ) ;
阿尔法( 3 + 5 , 7 ) ;
```

接下来，让我们通过添加proccall_statement 方法在我们的解析器中实现规则

```
def  proccall_statement ( self ):
    """proccall_statement : ID LPAREN (expr (COMMA expr)*)? RPAREN"""
    token  =  self . current_token

    proc_name  =  self 。current_token 。重视
    自我。吃（TokenType 。ID ）
    自我。吃（TokenType 。LPAREN ）
    actual_params  =  []
    ，如果 自己。current_token 。类型 !=  TokenType 。RPAREN :
        node  =  self 。expr ()
        actual_params 。追加（节点）

    而 自我。current_token 。类型 == 令牌类型。逗号：
        自我。吃（令牌类型。逗号）
        节点 = 自我。expr ()
        actual_params 。追加（节点）

    自我。吃（TokenType . RPAREN ）

    node  =  ProcedureCall (
        proc_name = proc_name ,
        actual_params = actual_params ,
        token = token ,
    )
    返回 节点
```

实现非常简单，并遵循语法规则：该方法解析过程调用并返回一个新的ProcedureCall AST 节点。

3. 我们需要对解析器进行的最后更改是：通过添加proccall_statement规则扩展语句语法规则并更新语句方法以调用proccall_statement 方法。

这是更新后的语句语法规则，其中包括proccall_statement 规则：

```
语句 ：  复合
           语句|  proccall_statement
           |  赋值语句
           |  空的
```

现在，我们手头有一个棘手的情况，我们有两个语法规则 - proccall_statement和assignment_statement - 它们以相同的标记开始，即ID标记。以下是他们完整的语法规则放在一起进行比较：

```
proccall_statement  :  ID  LPAREN  ( expr  ( COMMA  expr )*)？ 帕伦

assignment_statement  :  变量 ASSIGN  expr
变量：  ID
```

在这种情况下，您如何区分过程调用和赋值？它们都是语句，并且都以ID令牌开头。在下面的代码片段中，两个语句的ID令牌的值（词素）都是foo：

```
富() ;      { 过程调用}
foo  :=  5 ;   { 任务 }
```

解析器应该识别foo(); 上面作为一个过程调用和foo := 5; 作为任务。但是我们可以做些什么来帮助解析器区分过程调用和赋值呢？根据我们新的proccall_statement语法规则，过程调用以ID标记开始，后跟左括号。这就是我们将在解析器中依赖于区分过程调用和变量赋值的东西——ID 标记后的左括号的存在：

```
if  ( self . current_token . type  ==  TokenType . ID  and
    self . lexer . current_char  ==  '('
):
    node  =  self . proccall_statement ()
elif  self . current_token . type  ==  TokenType . ID :
    node  =  self . assignment_statement ( )
```

正如你在上面的代码中看到的，首先我们检查当前令牌是否是一个ID令牌，然后我们检查它是否跟有左括号。如果是，我们解析一个过程调用，否则我们解析一个赋值语句。

这是语句 方法的完整更新版本：

```
def 语句( self ):
    """
    语句 :
              Compound_statement
              |
proccall_statement | assignment_statement               | 空
    """
    if  self 。current_token 。类型 == 令牌类型。开始:
        node  =  self 。compound_statement （）
    的elif  （自我。current_token 。键入 ==  TokenType 。ID 和
          自我。词法分析器。current_char  ==  '('
    ):
        node  =  self 。proccall_statement ()
    elif  self 。current_token 。类型 == 令牌类型。ID :
        节点 = 自我。assignment_statement ()
    else :
        node  =  self 。空（）
    返回 节点
```

到现在为止还挺好。解析器现在可以解析过程调用。但是要记住的一件事是 Pascal 过程没有 return 语句，所以我们不能在表达式中使用过程调用。例如，如果Alpha是一个过程，则以下示例将不起作用：

```
x  :=  10  * 阿尔法( 3  +  5 ,  7 ) ;
```

这就是为什么我们只在statements方法中添加proccall_statement而没有其他地方。不用担心，在本系列的后面部分，我们将学习可以返回值并且还可以在表达式和赋值中使用的 Pascal 函数。

这些都是我们解析器的所有变化。接下来是语义分析器的变化。

为了支持过程调用，我们需要在语义分析器中进行的唯一更改是添加一个visit_ProcedureCall 方法：

```
DEF  visit_ProcedureCall （自， 节点）:
    用于 param_node 在 节点。实际参数:
        自我。访问（param_node ）
```

该方法所做的只是遍历传递给过程调用的实际参数列表，并依次访问每个参数节点。重要的是不要忘记访问每个参数节点，因为每个参数节点本身就是一个AST子树。

那很容易，不是吗？好的，现在继续更改解释器。

与语义分析器的变化相比，解释器的变化甚至更简单——我们只需要在解释器 类中添加一个空的visit_ProcedureCall方法：

```
def  visit_ProcedureCall ( self ,  node ):
    pass
```

有了上述所有更改，我们现在有了一个可以识别过程调用的解释器。我的意思是解释器可以解析过程调用并创建一个带有与这些过程调用相对应的ProcedureCall节点的AST。这是我们在文章开头看到的示例 Pascal 程序，我们希望在其上测试我们的解释器：

```
程序 主程序;

程序 Alpha （a ： 整数； b ： 整数）；
var x ： 整数;
开始
   x := ( a + b ) * 2 ;
结束;

开始 {主要}

   阿尔法( 3 + 5 , 7 ) ;  { 过程调用 }

结束。  { 主要的 }
```

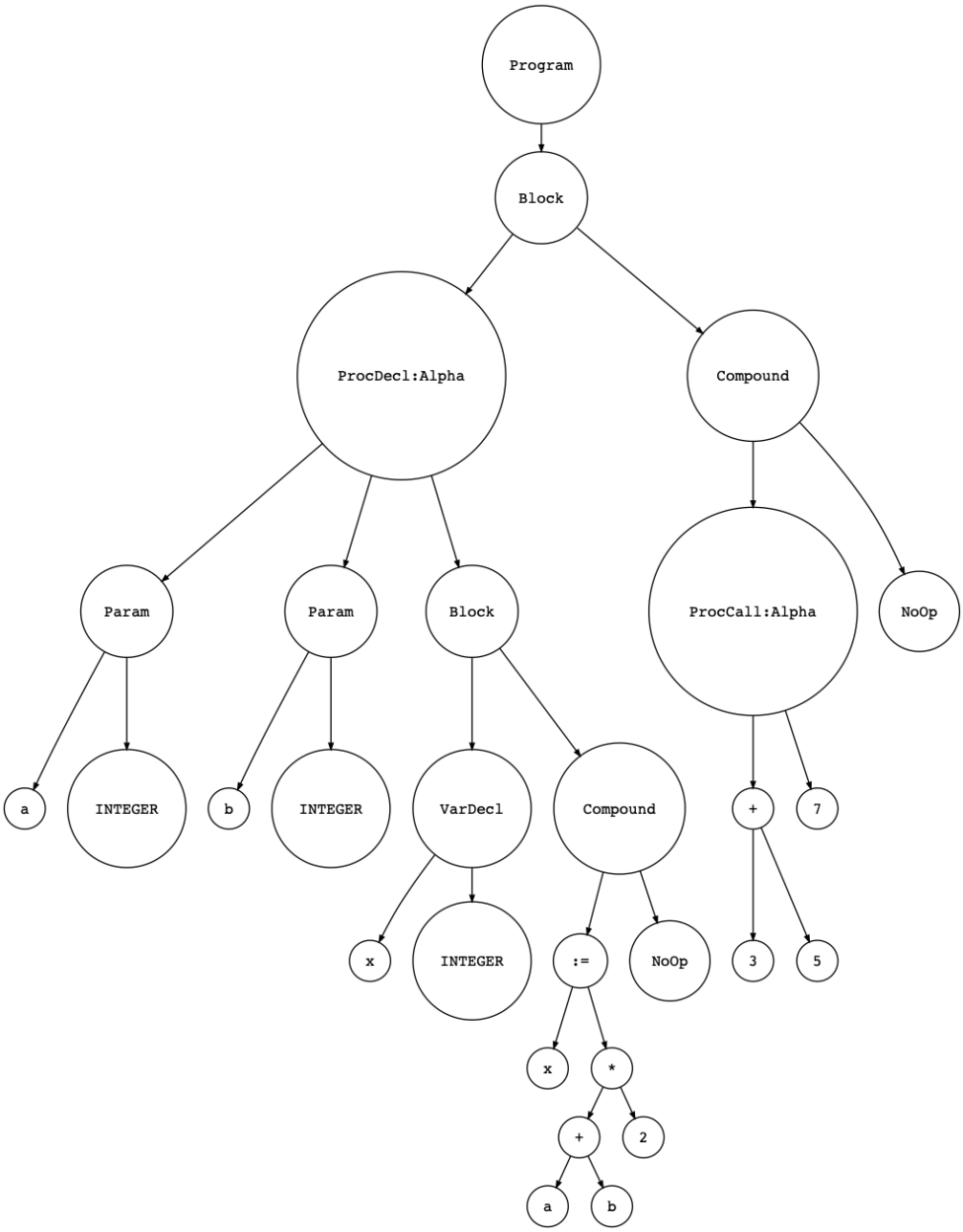
从GitHub (https://github.com/rspivak/lsbasi/tree/master/part16)下载上述程序或将代码保存到文件 part16.pas

亲自看看使用 part16.pas 作为输入文件运行我们更新的解释 (https://github.com/rspivak/lsbasi/tree/master/part16)器不会产生任何错误：

```
$ python spi.py part16.pas
$
```

到目前为止一切顺利，但没有输出也不是那么令人兴奋。:) 让我们稍微可视化一下并为上述程序生成一个AST，然后使用genastdot.py (https://github.com/rspivak/lsbasi/tree/master/part16/genastdot.py) 实用程序的更新版本来可视化AST： (https://github.com/rspivak/lsbasi/tree/master/part16/genastdot.py)

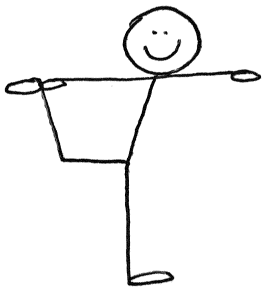
```
$ python genastdot.py part16.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```



这样更好。在上图中，您可以看到我们新的ProcCall AST节点标记为ProcCall:Alpha，用于Alpha(3 + 5, 7)过程调用。ProcCall:Alpha节点的两个子节点是传递给Alpha(3 + 5, 7)过程调用的参数3 + 5和7的子树。

好的，我们今天的目标已经完成：当遇到过程调用时，解析器构造一个带有ProcCall节点的AST用于过程调用，语义分析器和解释器在遍历AST时不会抛出任何错误。

现在，是时候锻炼了。



练习：向语义分析器添加一个检查，以验证传递给过程调用的参数（实际参数）的数量是否等于相应过程声明中定义的形式参数的数量。我们以文章前面使用的Alpha过程声明为例：

```
程序 Alpha （a ： 整数； b ： 整数）；
var x ： 整数；
开始
   x := ( a + b ) * 2 ;
结束;
```

上面过程声明中的形式参数的数量是两个（整数a和b）。如果您尝试使用多个参数而不是两个参数来调用该过程，则您的检查应该会抛出错误：

```
阿尔法() ;           { 0 个参数 -> 错误 }
Alpha ( 1 ) ;          { 1 个参数 -> 错误 }
Alpha ( 1 , 2 , 3 ) ;    { 3 个参数 -> 错误 }
```

您可以在GitHub 上 (https://github.com/rspivak/lsbasi/tree/master/part16)的文件解决方案.txt 中 (https://github.com/rspivak/lsbasi/tree/master/part16)找到该练习的解决方案，但在查看文件之前先尝试制定您自己的解决方案。

这就是今天的全部内容。在下一篇文章中，我们将开始学习如何解释过程调用。我们将介绍调用堆栈和激活记录等主题。这将是一次疯狂的旅程:) 所以请继续关注我们，下次再见！

用于准备本文的资源（一些链接是附属链接）：

1. 语言实现模式：创建您自己的特定领域和通用编程语言（实用程序员） (https://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=5d5ca8c07bff5452ea443d8319e7703d)
2. 编写编译器和解释器：一种软件工程方法 (https://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=542d1267e34a529e0f69027af20e27f3)
3. 免费帕斯卡参考指南 (https://www.freepascal.org/docs-html/current/ref/ref.html)

如果您想在收件箱中获取我的最新文章，请在下方输入您的电子邮件地址，然后单击“获取更新”！

输入您的名字 *

输入您最好的电子邮件 *

获取更新！

本系列所有文章：

- 让我们构建一个简单的解释器。第1部分。 (/lsbasi-part1/)
- 让我们构建一个简单的解释器。第2部分。 (/lsbasi-part2/)
- 让我们构建一个简单的解释器。第 3 部分。 (/lsbasi-part3/)
- 让我们构建一个简单的解释器。第 4 部分。 (/lsbasi-part4/)
- 让我们构建一个简单的解释器。第 5 部分。 (/lsbasi-part5/)
- 让我们构建一个简单的解释器。第 6 部分。 (/lsbasi-part6/)
- 让我们构建一个简单的解释器。第 7 部分：抽象语法树 (/lsbasi-part7/)

- 让我们构建一个简单的解释器。第 8 部分。 (/lsbasi-part8/)
- 让我们构建一个简单的解释器。第 9 部分。 (/lsbasi-part9/)
- 让我们构建一个简单的解释器。第 10 部分。 (/lsbasi-part10/)
- 让我们构建一个简单的解释器。第 11 部分。 (/lsbasi-part11/)
- 让我们构建一个简单的解释器。第 12 部分。 (/lsbasi-part12/)
- 让我们构建一个简单的解释器。第 13 部分：语义分析 (/lsbasi-part13/)
- 让我们构建一个简单的解释器。第 14 部分：嵌套作用域和源到源编译器 (/lsbasi-part14/)
- 让我们构建一个简单的解释器。第 15 部分。 (/lsbasi-part15/)
- 让我们构建一个简单的解释器。第 16 部分：识别过程调用 (/lsbasi-part16/)
- 让我们构建一个简单的解释器。第 17 部分：调用堆栈和激活记录 (/lsbasi-part17/)
- 让我们构建一个简单的解释器。第 18 部分：执行过程调用 (/lsbasi-part18/)
- 让我们构建一个简单的解释器。第 19 部分：嵌套过程调用 (/lsbasi-part19/)

注释

ALSO ON **RUSLAN'S BLOG**

Let's Build A Simple Interpreter. Part 3.

6 years ago • 15 comments

I woke up this morning and I thought to myself: "Why do we find it so difficult to …

EOF is not a character

2 years ago • 16 comments

I was reading Computer Systems: A Programmer's Perspective the other day …

Let's Build A Simple Interpreter. Part 2.

6 years ago • 16 comments

In their amazing book "The 5 Elements of Effective Thinking" the authors …

Let's B Server.

7 years a

Out for a woman c construc

7 Comments **Ruslan's Blog** **Disqus' Privacy Policy** **Login**

Recommend 2 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS**

Name

Robert • 2 years ago

I love this series! Thank you. I'm only at the beginning of this one but I've found some errors in proccall_statement.

1) Your definition

proccall_statement : ID LPAREN expr? (COMMA expr)* RPAREN

means that this would be valid

Alpha(, 1)

I believe it should be

proccall_statement : ID LPAREN (expr (COMMA expr)*)? RPAREN

2) The diagram for proccall_statement is linear whereas the branch for the bottom expr should loop back

Keep up the great articles. I can't wait to finish this one.

1 • Reply • Share ›

rspivak Mod Robert • 2 years ago

Good catch! Thanks, Robert. I've updated the article.

1 • Reply • Share ›

Hermes Passer • 2 years ago

loving this series

• Reply • Share ›

paulnicholls • 2 years ago

Thanks for sharing all your knowledge Ruslan!! I've been following along and made my own Pascal parser -> AST tree using the community version of Delphi! I'm looking forward to the next installment :)

• Reply • Share ›

rspivak Mod ➚ paulnicholls • 2 years ago

That's great. :) Good luck with your project!

• Reply • Share ›

paulnicholls ➚ rspivak • 2 years ago • edited

Thanks I have previously read Jack Crenshaw's "let's build a compiler" series which is excellent for direct code generation with no optimisations. Using his series, I have coded a Pascal to 6502 assembly compiler called Pas6502 on Bitbucket.

https://bitbucket.org/paul_...

I now wanted to generate an ast from the parser instead of direct code and I stumbled across your series which has helped immensely!!

I'm now modifying it (my local copy) to generate the ast (done) and I will now start code generation from walking the ast

• Reply • Share ›

Chris Evans • 2 years ago

You're awesome!

由Disqus提供支持的评论 (http://disqus.com)

社会的

- github (https://github.com/rspivak/)
- 推特 (https://twitter.com/rspivak)
- 链接 (https://linkedin.com/in/ruslanspivak/)

热门帖子

- 让我们构建一个 Web 服务器。第1部分。 (https://ruslanspivak.com/lsbaws-part1/)
- 让我们构建一个简单的解释器。第1部分。 (https://ruslanspivak.com/lsbasi-part1/)
- 让我们构建一个 Web 服务器。第2部分。 (https://ruslanspivak.com/lsbaws-part2/)
- 让我们构建一个 Web 服务器。第 3 部分。 (https://ruslanspivak.com/lsbaws-part3/)
- 让我们构建一个简单的解释器。第2部分。 (https://ruslanspivak.com/lsbasi-part2/)

免责声明

这个网站上的一些链接有我的亚马逊推荐 ID，它为我提供了每次销售的小额佣金。感谢您的支持。

↑ 回到顶部