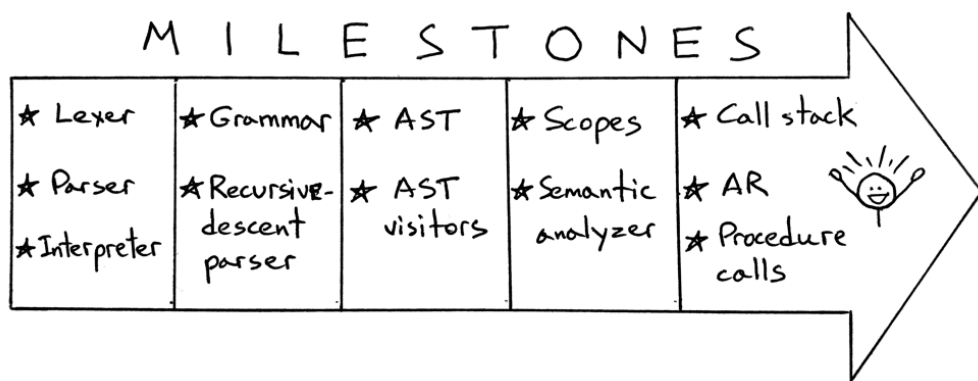


# 让我们构建一个简单的解释器。第 18 部分：执行过程调用 (<https://ruslanspivak.com/lsbasi-part18/>)

日期 2020 年 2 月 20 日，星期四

“尽你所能，直到你知道得更好。然后当你知道得更好时，做得更好。” ——玛雅·安吉洛

今天对我们来说是一个巨大的里程碑！因为今天我们将扩展我们的解释器来执行过程调用。如果这不令人兴奋，我不知道是什么。:)



你准备好了吗？让我们开始吧！

这是我们将在本文中重点关注的示例程序：

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 }

它有一个过程声明和一个过程调用。我们今天将把重点限制在只能访问其参数和局部变量的过程上。我们将在接下来的两篇文章中介绍嵌套过程调用和访问非局部变量。

让我们描述一个我们的解释器需要实现的算法，以便能够执行上面程序中的Alpha(3 + 5, 7)过程调用。

这是执行过程调用的算法，一步一步：

1. 创建激活记录
2. 在活动记录中保存过程参数（实际参数）
3. 将激活记录推入调用堆栈

#### 4. 执行程序体

#### 5. 从堆栈中弹出激活记录

我们解释器中的过程调用由visit\_ProcedureCall方法处理。该方法目前为空：

```
类 解释器 (NodeVisitor) :
    ...

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

让我们回顾一下算法中的每一步，并为visit\_ProcedureCall方法编写代码以执行过程调用。

让我们开始吧！

##### 步骤 1. 创建激活记录

如果您还记得上一篇文章 (<https://ruslanspivak.com/lbasi-part17>)，活动记录 ( AR )是一个类似字典的对象，用于维护有关当前正在执行的过程或函数调用以及程序本身的信息。例如，过程的活动记录包含其形式参数的当前值和其局部变量的当前值。因此，要存储过程的参数和局部变量，我们需要先创建一个AR。回想一下 ActivationRecord构造函数接受 3 个参数：name、type和nesting\_level。这是我们在创建一个时需要传递给构造函数的内容过程调用的AR：

- 我们需要将过程的名称作为名称参数传递给构造函数
- 我们还需要指定程序的类型中的AR
- 并且我们需要传递2作为过程调用的nesting\_level，因为程序的嵌套级别设置为1（你可以在解释器的visit\_Program方法中看到）

在我们扩展visit\_ProcedureCall方法为过程调用创建激活记录之前，我们需要将PROCEDURE类型添加到ARType枚举中。让我们先这样做：

```
类 ARType (枚举) :
    PROGRAM    = 'PROGRAM'
    PROCEDURE  = 'PROCEDURE'
```

现在，让我们更新visit\_ProcedureCall方法，以使用我们前面在文本中描述的适当参数来创建激活记录：

```
def visit_ProcedureCall ( self , node ) :
    proc_name = node 。 进程名称

    ar = ActivationRecord (
        name = proc_name ,
        type = ARType . PROCEDURE ,
        nesting_level = 2 ,
    )
```

一旦我们弄清楚将什么作为参数传递给ActivationRecord构造函数，编写代码来创建激活记录就很容易了。

##### 步骤 2. 在激活记录中保存过程参数

ASIDE：形式参数是出现在过程声明中的参数。实际参数（也称为参数）是在特定过程调用中传递给过程的不同变量和表达式。

以下是描述解释器在活动记录中保存过程参数所需采取的高级操作的步骤列表：

- 获取过程的形参列表
- 获取过程的实际参数（arguments）列表
- 对于每个形参，获取对应的实参，以形参名称为键，实参（实参）求值后作为值保存在过程的激活记录中

如果我们有以下过程声明和过程调用：

```
程序 Alpha ( a : 整数; b : 整数 );

阿尔法( 3 + 5 , 7 );
```

那么上面三个步骤执行完后，程序的AR内容应该是这样的：

```
2 : 程序 阿尔法
  a           : 8
  b           : 7
```

下面是实现上述步骤的代码：

```
proc_symbol = 节点。proc_symbol

正式参数 = proc_symbol 。formal_params
actual_params = node 。实际参数

对于 param_symbol , argument_node 在 拉链 (formal_params , actual_params ) :
    AR [ param_symbol 。名称 ] = 自我。访问 (argument_node )
```

让我们仔细看看步骤和代码。

a) 首先，我们需要得到一个过程的形参列表。我们可以从哪里得到它们？它们在语义分析阶段创建的相应过程符号中可用。为了唤起你的记忆，这里是ProcedureSymbol 类的定义：

```
class Symbol :
    def __init__ ( self , name , type = None ):
        self 。姓名 = 姓名
        自我。类型 = 类型

class ProcedureSymbol ( Symbol ):
    def __init__ ( self , name , formal_params = None ):
        super () 。__init__ ( name )
        # VarSymbol 对象列表
        self . formal_params = [] , 如果 formal_params 是 无 其他 formal_params
```

这是全局范围（程序级别）的内容，它显示了Alpha过程符号及其形式参数的字符串表示形式：

```
范围（范围符号表）
=====
范围名称：全球
范围级别：1
封闭范围：无
范围（Scoped Symbol table）内容
-----
整数：<BuiltinTypeSymbol(name='INTEGER')>
  真实：<BuiltinTypeSymbol(name='REAL')>
  Alpha：<ProcedureSymbol(name=Alpha, parameters=[ <VarSymbol(name='a', type='INTEGER')> , <VarSymbol(name='b'
```

好的，我们现在知道从哪里获取形式参数。我们如何从ProcedureCall AST 节点变量获得过程符号？让我们看一下到目前为止我们编写的visit\_ProcedureCall方法代码：

```
def visit_ProcedureCall ( self , node ):
    proc_name = node 。 进程名称

    ar = ActivationRecord (
        name = proc_name ,
        type = ARType . PROCEDURE ,
        nesting_level = 2 ,
    )
```

我们可以通过在上面的代码中添加以下语句来访问过程符号：

```
proc_symbol = 节点。proc_symbol
```

但是如果查看上一篇文章 (<https://ruslanspivak.com/lbasi-part17>)中的ProcedureCall类的定义，可以看到该类没有proc\_symbol作为成员：<https://ruslanspivak.com/lbasi-part17>)

```
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
```

让我们解决这个问题并扩展ProcedureCall类以具有proc\_symbol 字段：

```
class ProcedureCall(AST):
    def __init__(self, proc_name, actual_params, token):
        self.proc_name = proc_name
        self.actual_params = actual_params # a list of AST nodes
        self.token = token
        # a reference to procedure declaration symbol
        self.proc_symbol = None
```

那很简单。现在，我们应该在哪里设置proc\_symbol以便它具有解释阶段的正确值（对相应过程符号的引用）？正如我之前提到的，过程符号是在语义分析阶段创建的。我们可以在语义分析器的visit\_ProcedureCall 方法完成节点遍历期间将其存储在ProcedureCall AST节点中。

这是原始方法：

```
类 语义分析器 (NodeVisitor ) :
    ...

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

因为在语义分析器中遍历AST树时我们可以访问当前范围，所以我们可以通过过程名称查找过程符号，然后将过程符号存储在ProcedureCall AST节点的proc\_symbol变量中。我们开工吧：

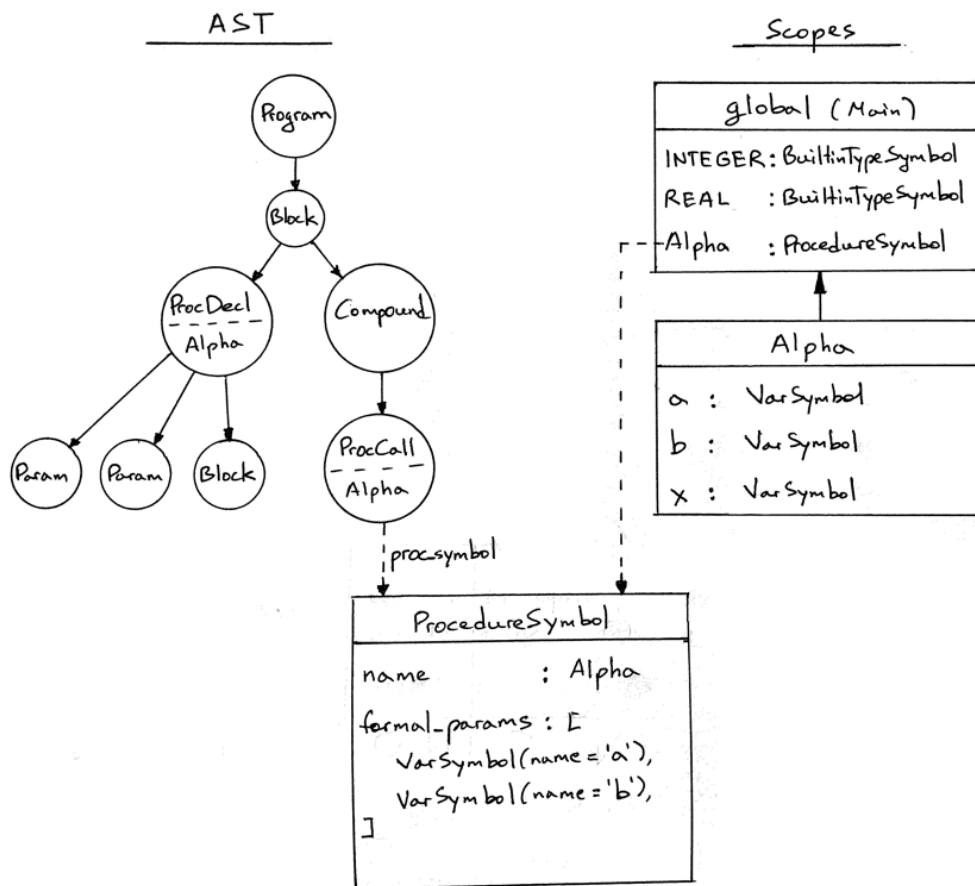
```
类 语义分析器 (NodeVisitor ) :
    ...

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

        proc_symbol = self 。 current_scope 。 查找 ( 节点。proc_name中 )
        # 执行过程调用时被解释器访问
        节点。proc_symbol = proc_symbol
```

在上面的代码中，我们简单地将过程名称解析为其过程符号，该过程符号存储在一个范围符号表中（在我们的例子中，准确地说是全局范围），然后将过程符号分配给proc\_symbol字段所述的ProcedureCall AST 节点。

对于我们的示例程序，在语义分析阶段和上述操作之后，AST树将在全局范围内有一个到Alpha过程符号的链接：



如上图所示，此设置允许我们从解释器的visit\_ProcedureCall方法中获取过程的形式参数- 在评估ProcedureCall节点时 - 只需访问存储在ProcedureCall AST 节点中的proc\_symbol变量的formal\_params字段：

```
proc_symbol = 节点.proc_symbol

proc_symbol . formal_params # 又名参数
```

b) 在我们得到形参列表之后，我们需要得到一个过程的实参（arguments）列表。获取参数列表很容易，因为它们很容易从ProcedureCall AST 节点 本身获得：

```
节点.actual_params # 又名参数
```

c) 最后一步。对于每个形参，我们需要获取对应的实参，并以形参名称为键，实参（实参）求值后作为值保存在过程的激活记录中

让我们看一下使用 Python zip() (<https://docs.python.org/3/library/functions.html#zip>) 函数构建键值对的代码：

```
proc_symbol = 节点.proc_symbol

正式参数 = proc_symbol . formal_params
actual_params = 节点 . 实际参数

对于 param_symbol , argument_node 在 拉链 (formal_params , actual_params) :
    AR [ param_symbol . 名称 ] = 自我.访问 (argument_node )
```

一旦你知道 Python `zip()` (<https://docs.python.org/3/library/functions.html#zip>) 函数是如何工作的，上面的 for 循环应该很容易理解。这是 `zip()` (<https://docs.python.org/3/library/functions.html#zip>) 函数的 Python shell 演示：

```
>>> formal_params = ['a', 'b', 'c']
>>> 实际参数 = [1, 2, 3]
>>>
>>> 压缩 = zip(formal_params, actual_params)
>>>
>>> 列表（压缩）
[('a', 1), ('b', 2), ('c', 3)]
```

将键值对存储在活动记录中的语句非常简单：

```
ar [ param_symbol . 名称 ] = 自我。访问 ( argument_node )
```

键是形式参数的名称，值是传递给过程调用的参数的计算值。

这是解释器的 `visit_ProcedureCall` 方法，其中包含我们迄今为止所做的所有修改：

```
类 解释器 (NodeVisitor) :
    ...

    def visit_ProcedureCall ( self , node ) :
        proc_name = node . 进程名称

        ar = ActivationRecord (
            name = proc_name ,
            type = ARType . PROCEDURE ,
            nesting_level = 2 ,
        )

        proc_symbol = 节点 . proc_symbol

        正式参数 = proc_symbol . formal_params
        actual_params = node . 实际参数

        对于 param_symbol , argument_node 在 拉链 ( formal_params , actual_params ) :
            AR [ param_symbol . 名称 ] = 自我。访问 ( argument_node )
```

### 步骤 3. 将激活记录推入调用堆栈

在我们创建了 AR 并将所有过程的参数放入 AR 之后，我们需要将 AR 压入堆栈。做起来超级容易。我们只需要添加一行代码：

```
自我。调用堆栈。推( ar )
```

请记住：当前正在执行的过程的 AR 始终位于堆栈顶部。通过这种方式，当前正在执行的过程可以轻松访问其参数和局部变量。这是更新后的 `visit_ProcedureCall` 方法：

```
def visit_ProcedureCall ( self , node ):
    proc_name = node 。 进程名称

    ar = ActivationRecord (
        name = proc_name ,
        type = ARTYPE . PROCEDURE ,
        nesting_level = 2 ,
    )

    proc_symbol = 节点。proc_symbol

    正式参数 = proc_symbol 。 formal_params
    actual_params = node 。 实际参数

    对于 param_symbol , argument_node 在 拉链 ( formal_params , actual_params ) :
        AR [ param_symbol 。 名称 ] = 自我。访问 ( argument_node )

    自我。调用堆栈。推( ar )
```

#### 步骤 4. 执行过程的主体

现在一切都已经设置好了，让我们执行过程的主体。唯一的问题是ProcedureCall AST 节点和过程符号proc\_symbol 都不知道有关相应过程声明的主体的任何信息。

我们如何在执行过程调用期间访问过程声明的主体？换句话说，在解释阶段遍历AST树并访问ProcedureCall AST节点时，我们需要访问对应ProcedureDecl节点的block\_node变量。所述block\_node变量保存到一个参考AST子树表示该过程的主体。我们如何从Interpreter类的visit\_ProcedureCall方法访问该变量？让我们考虑一下。

我们已经可以访问包含有关过程声明的信息的过程符号，比如过程的形式参数，所以让我们找到一种方法来在过程符号本身中存储对block\_node的引用。正确的做法是语义分析器的visit\_ProcedureDecl方法。在这个方法中，我们可以访问过程符号和过程主体，即指向过程主体的AST 子树的ProcedureDecl AST节点的block\_node字段。

我们有一个过程符号，我们有一个block\_node。让我们存储的指针block\_node在block\_ast基于场proc\_symbol：

```
类 语义分析器 (NodeVisitor ) :

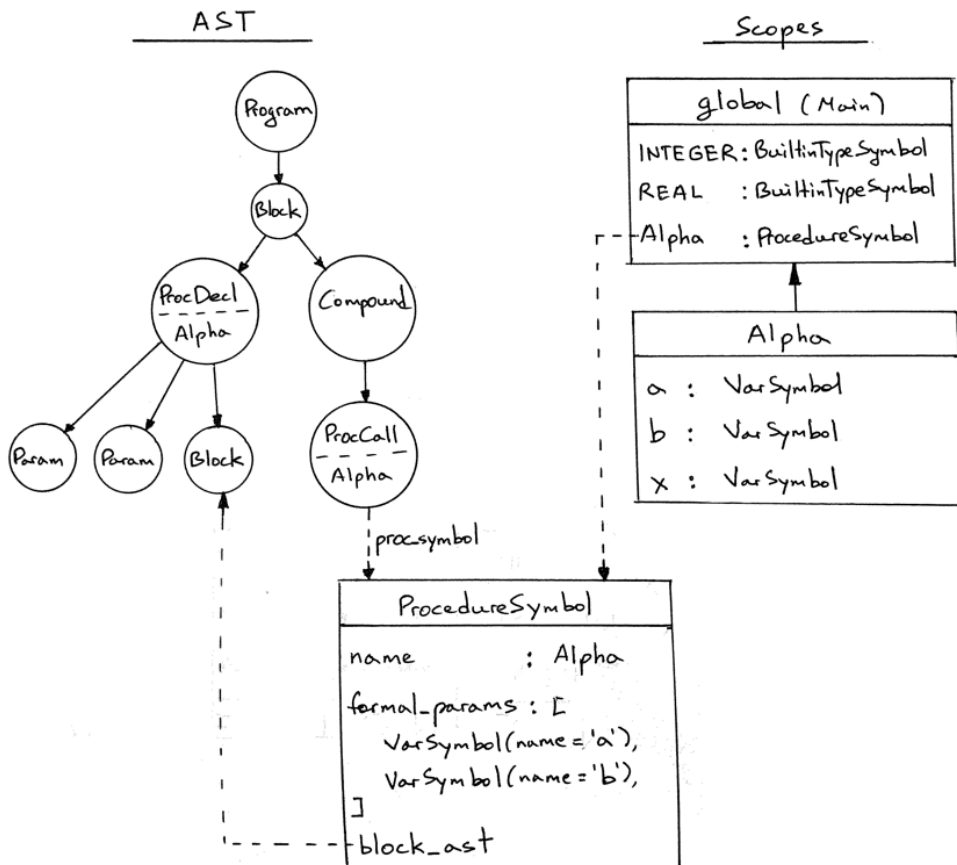
    def visit_ProcedureDecl ( self , node ):
        proc_name = node 。 proc_name
        proc_symbol = ProcedureSymbol ( proc_name )
        ...
        self 。 日志 ( f 'LEAVE 范围: {proc_name}' )

        # 在执行过程调用
        proc_symbol时由解释器访问。block_ast = 节点。块节点
```

为了明确起见，我们还扩展了ProcedureSymbol类并向其中添加了block\_ast字段：

```
class ProcedureSymbol ( Symbol ):
    def __init__ ( self , name , formal_params = None ):
        ...
        # 对过程主体 (AST 子树) 的引用
        self 。 block_ast = 无
```

在下图中，您可以看到扩展的ProcedureSymbol实例，该实例存储对相应过程主体（AST 中的Block节点）的引用：



有了上述所有内容，在过程调用中执行过程主体变得像访问过程声明的Block AST节点一样简单，该节点可通过过程的proc\_symbol的block\_ast字段访问：

自我。访问（proc\_symbol . block\_ast ）

这是Interpreter 类的完全更新的visit\_ProcedureCall方法：

```
def visit_ProcedureCall ( self , node ):
    proc_name = node 。 进程名称

    ar = ActivationRecord (
        name = proc_name ,
        type = ARTYPE . PROCEDURE ,
        nesting_level = 2 ,
    )

    proc_symbol = 节点.proc_symbol

    正式参数 = proc_symbol 。 formal_params
    actual_params = node 。 实际参数

    对于 param_symbol , argument_node 在 拉链 (formal_params , actual_params ) :
        AR [ param_symbol 。 名称 ] = 自我。访问 ( argument_node )

    自我。调用堆栈。推( ar )

    # 评估过程体
    self . 访问 ( proc_symbol . block_ast )
```

如果您还记得上一篇文章 (<https://ruslanspivak.com/lbasi-part17>)，visit\_Assignment和visit\_Var方法使用调用堆栈顶部的AR来访问和存储变量：



```
def visit_Assign ( self , node ):
    var_name = node 。离开了。值
    var_value = self 。访问 (节点。右)

    ar = 自我。调用堆栈。peek ()
    ar [ var_name ] = var_value

def visit_Var ( self , node ):
    var_name = node 。价值

    ar = 自我。调用堆栈。peek ()
    var_value = ar 。获取( var_name )

    返回 变量值
```

这些方法保持不变。在解释过程的主体时，这些方法将存储和访问来自当前执行过程的AR的值，该AR将位于堆栈的顶部。我们很快就会看到它们是如何配合并协同工作的。

### 步骤 5. 从堆栈中弹出激活记录

在我们完成对过程体的评估之后，我们不再需要过程的AR，所以我们在离开visit\_ProcedureCall方法之前将它从调用堆栈中弹出。请记住，调用堆栈的顶部包含当前正在执行的过程、函数或程序的AR，因此一旦我们完成对这些例程之一的评估，我们需要使用调用堆栈的pop将它们各自的AR从调用堆栈中弹出() 方法：

自我。调用堆栈。流行()

让我们把它们放在一起，并在visit\_ProcedureCall方法中添加一些日志记录，以便在将过程的AR推入调用堆栈之后和将其从堆栈中弹出之前记录调用堆栈的内容：

```
def visit_ProcedureCall ( self , node ):
    proc_name = node 。进程名称

    ar = ActivationRecord (
        name = proc_name ,
        type = ARType . PROCEDURE ,
        nesting_level = 2 ,
    )

    proc_symbol = 节点。proc_symbol

    正式参数 = proc_symbol 。formal_params
    actual_params = node 。实际参数

    对于 param_symbol , argument_node 在 拉链 ( formal_params , actual_params ) :
        AR [ param_symbol 。名称 ] = 自我。访问 ( argument_node )

    自我。调用堆栈。推( ar )

    自我。日志( f 'ENTER: PROCEDURE {proc_name}' )
    self 。日志( str ( self . call_stack ))

    # 评估过程体
    self 。访问 ( proc_symbol . block_ast )

    自我。日志( f 'LEAVE: PROCEDURE {proc_name}' )
    self 。日志( str ( self . call_stack ))

    自我。调用堆栈。流行()
```

让我们试一试我们修改过的解释器，看看它是如何执行过程调用的。从[GitHub](https://github.com/rspivak/lspivak/tree/master/part18)

(<https://github.com/rspivak/lspivak/tree/master/part18>)下载以下示例程序或将其保存为part18.pas

(<https://github.com/rspivak/lbasi/blob/master/part18/part18.pas>):

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/lbasi/tree/master/part18/>)下载解释器文件spi.py (<https://github.com/rspivak/lbasi/blob/master/part18/spi.py>)并使用以下参数在命令行上 (<https://github.com/rspivak/lbasi/tree/master/part18/>)运行它:

```
$ python spi.py part18.pas --stack
```

输入: 程序主

调用堆栈

1 : 程序主

输入: 程序阿尔法

调用堆栈

2: 程序阿尔法

a : 8

b : 7

1 : PROGRAM Main

离开: 程序阿尔法

调用堆栈

2: 程序阿尔法

a : 8

b : 7

x : 30

1 : PROGRAM Main

离开: 程序主要

调用堆栈

1 : 程序主

到现在为止还挺好。让我们仔细查看输出并检查程序和过程执行期间调用堆栈的内容。

## 1.解释器先打印

```
ENTER : PROGRAM Main
```

```
CALL STACK
```

```
1 : PROGRAM Main
```

在执行程序主体之前访问Program AST节点时。此时调用堆栈有一个激活记录。此活动记录位于调用堆栈的顶部，用于存储全局变量。因为我们的示例程序中没有任何全局变量，所以活动记录中没有任何内容。

## 2.接下来，解释器打印

```

ENTER : PROCEDURE Alpha
CALL STACK
2 : PROCEDURE Alpha
  a      : 8
  b      : 7
1 : PROGRAM Main

```

当它访问Alpha(3 + 5, 7)过程调用的ProcedureCall AST节点时。此时Alpha过程的主体还没有被评估，调用堆栈有两个激活记录：一个用于堆栈底部的Main程序（嵌套级别 1），另一个用于Alpha过程调用，在栈顶（嵌套级别 2）。堆栈顶部的AR仅保存过程参数a和b的值；有本地变量没有值X在AR 因为程序的主体还没有被评估。

### 3.接下来，解释器打印

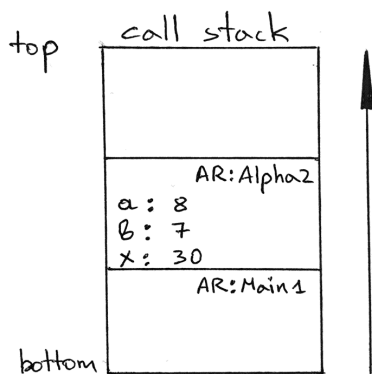
```

离开： 程序 阿尔法
呼叫 堆栈
2 : 程序 阿尔法
  a      : 8
  b      : 7
  x      : 30
1 : 程序 主

```

当它即将离开Alpha(3 + 5, 7)过程调用的ProcedureCall AST节点时，但在为Alpha 过程弹出AR之前。

从上面的输出中，您可以看到除了过程参数之外，当前正在执行的过程Alpha的AR现在还包含对局部变量x的赋值结果，即执行 $x := (a + b) * 2$ ; 程序体中的语句。此时调用堆栈在视觉上看起来是这样的：



### 4. 最后解释器打印

```

离开： PROGRAM Main
CALL STACK
1 : PROGRAM Main

```

当它离开Program AST节点但在它弹出主程序的AR之前。正如你所看到的，主程序启动记录是唯一的AR在栈中离开，因为AR为阿尔法过程调用得到弹出堆栈早些时候，右完成执行之前的Alpha过程调用。

就是这样。我们的解释器成功地执行了一个过程调用。如果你已经到了这一步，恭喜你！



这对我们来说是一个巨大的里程碑。现在您知道如何执行过程调用了。如果您已经等待这篇文章很久了，感谢您的耐心等待。

这就是今天的全部内容。在下一篇文章中，我们将扩展当前材料并讨论执行嵌套过程调用。所以请继续关注我们下期再见！

用于准备本文的资源（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](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](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. 编程语言语用学，第四版 ([https://www.amazon.com/gp/product/0124104096/ref=as\\_li\\_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0124104096&linkCode=as2&tag=russblo0b-20&linkId=8db1da254b12fe6da1379957dda717fc](https://www.amazon.com/gp/product/0124104096/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0124104096&linkCode=as2&tag=russblo0b-20&linkId=8db1da254b12fe6da1379957dda717fc))

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 2.

6 years ago • 16 comments

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

Let's Build A Simple Interpreter. Part 1.

6 years ago • 61 comments

"If you don't know how compilers work, then you don't know how ...

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 19: ...

2 years ago • 24 comments

What I cannot create, I do not understand. — Richard Feynman

Let's Interp

6 years


As I pr today the ce

Recommend

Tweet

Share

Sort by Best




Join the discussion...

LOG IN WITH


OR SIGN UP WITH DISQUS ?

Name

- 


gabodev • 2 years ago

I was waiting for this a lot. I love it, thank you very much.

1 ^ | v • Reply • Share ›
- 


rspivak Mod ➔ gabodev • 2 years ago

You're welcome. :) I'm glad you liked it.

^ | v • Reply • Share ›
- 


MikolajM • 2 years ago

I need to thank you! I am currently writing my bachelor thesis and need to develop and interpret really simple machine language and your series helped me a ton! Thank you for sharing this knowledge and spending your time to teach us :)

^ | v • Reply • Share ›
- 

Rajesh Pillai • 2 years ago


Your articles are one if the best 👍

^ | v • Reply • Share ›
- 

rickard1117@163.com • 2 years ago

Thank you very much , this is the BEST tutorial of compiler I've ever found. Now I'm implementing a Pacal Interpreter in C++ : <https://github.com/rickard1117/spi4c>


^ | v • Reply • Share ›



rspivak Mod ➔ rickard1117@163.com • 2 years ago

Thanks. :) Good luck with your project!

^ | v • Reply • Share ›




Guga Loks • 2 years ago

Well... will we have a 19 part? Haha

Good job.

^ | v • Reply • Share ›



Lahsiv • 2 years ago

Was waiting for the next article in the series for quite some time. Was long overdue :-). Need to re-read previous posts to recall the forgotten bits.

^ | v • Reply • Share ›
- Subscribe

Add Disqus to your siteAdd DisqusAdd

Do Not Sell My Data
- 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
- <https://ruslanspivak.com/lsbasi-part18/>

14/15

让我们构建一个 Web 服务器。第1部分。(<https://ruslanspivak.com/lbaws-part1/>)

让我们构建一个简单的解释器。第1部分。(<https://ruslanspivak.com/lbasi-part1/>)

让我们构建一个 Web 服务器。第2部分。(<https://ruslanspivak.com/lbaws-part2/>)

让我们构建一个 Web 服务器。第 3 部分。(<https://ruslanspivak.com/lbaws-part3/>)

让我们构建一个简单的解释器。第2部分。(<https://ruslanspivak.com/lbasi-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。