

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 (<https://ruslanspivak.com/lbasi-part17/>)

日期 2019 年 8 月 28 日，星期三

“你可能需要不止一次地战斗才能赢得它。” - 玛格丽特·撒切尔

1968 年墨西哥城夏季奥运会期间，一位名叫约翰·斯蒂芬·阿赫瓦里 (John Stephen Akhwari) 的马拉松选手发现自己与东非的祖国坦桑尼亚相距数千英里。在墨西哥城的高海拔地区跑马拉松时，他被其他争夺位置的运动员撞倒在地，膝盖严重受伤并导致脱臼。接受医疗救治后，他没有因伤势过重而退出比赛，而是站起来继续比赛。

埃塞俄比亚的 Mamo Wolde 在 2:20:26 进入比赛时，以第一名冲过终点线。一个多小时后的 3:25:27，太阳落山后，Akhwari 蹒跚着，一条血淋淋的腿和他的绷带在风中摇晃着，最后越过终点线。

当一小群人看到 Akhwari 越线时，他们难以置信地为他欢呼，剩下的几个记者冲上赛道问他为什么受伤了还要继续比赛。他的回应载入史册：“我的国家没有让我跑 5,000 英里来开始比赛。他们让我跑了 5,000 英里来完成比赛。”

这个故事从那时起激励了许多运动员和非运动员。此时您可能会想，“太好了，这是一个鼓舞人心的故事，但这与我有什么关系？”对你我来说的主要信息是：“继续前进！”这是一个很长一段时间的长系列，有时可能会让人望而生畏，但我们正在接近该系列的一个重要里程碑，所以我们需要继续前进。



好的，让我们开始吧！

我们今天有几个目标：

1. 实现可以支持程序、过程调用和函数调用的新内存系统。
2. 用新的内存系统替换解释器当前的内存系统，由GLOBAL_MEMORY字典表示。

让我们从回答以下问题开始：

1. 什么是记忆系统？
2. 为什么我们需要一个新的记忆系统？

3. 新的记忆系统是什么样的？

4. 为什么我们要替换GLOBAL_MEMORY 字典？

1.什么是记忆系统？

简单的说，就是在内存中存储和访问数据的系统。在硬件级别，它是物理内存（RAM），其中值存储在特定的物理地址。在解释器级别，因为我们的解释器根据变量名而不是物理地址来存储值，所以我们用一个将名称映射到值的字典来表示内存。这是一个简单的演示，我们通过变量名称y存储 7 的值，然后立即访问与名称y关联的值：

```
>>> GLOBAL_MEMORY = {}
>>>
>>> GLOBAL_MEMORY['y'] = 7 # 按名称存储值
>>>
>>> GLOBAL_MEMORY['y'] # 按名称访问值
7
>>>
```

一段时间以来，我们一直在使用这种字典方法来表示全局内存。我们一直在使用GLOBAL_MEMORY字典在PROGRAM级别（全局级别）存储和访问变量。以下是与“内存”创建相关的解释器部分，处理内存中变量的值分配以及通过名称访问值：

```
class Interpreter ( NodeVisitor ):
    def __init__ ( self , tree ):
        self . 树 = 树
        自我。GLOBAL_MEMORY = {}

    def visit_Assign ( self , node ):
        var_name = node . 离开了。值
        var_value = self . 访问 ( 节点。右 )
        自我。GLOBAL_MEMORY [ var_name ] = var_value

    def visit_Var ( self , node ):
        var_name = node . 值
        var_value = self . 全局内存。get ( var_name )
        返回 var_value
```

既然我们已经描述了我们当前如何在解释器中表示内存，让我们找出下一个问题的答案。

2.为什么我们的解释器需要一个新的记忆系统？

事实证明，只有一个字典来表示全局内存不足以支持过程和函数调用，包括递归调用。

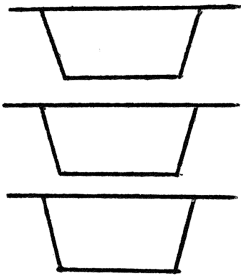
为了支持嵌套调用，以及嵌套调用的特殊情况，递归调用，我们需要多个字典来存储有关每个过程和函数调用的信息。我们需要那些以特定方式组织的词典。这就是我们需要一个新的记忆系统的原因。拥有这个内存系统是执行过程调用的垫脚石，我们将在以后的文章中实现。

3.新的内存系统是什么样的？

在其核心，新的内存系统是一个堆栈数据结构，将类似字典的对象作为其元素。该堆栈称为“**调用堆栈**”，因为它用于跟踪当前正在执行的过程/函数调用。该调用栈也被称为运行时堆栈，执行堆栈，程序堆栈，或者只是“栈”。调用堆栈保存的类似字典的对象称为**活动记录**。你可能知道它们的另一个名字：“堆栈帧”，或者只是“帧”。

让我们更详细地了解调用堆栈和激活记录。

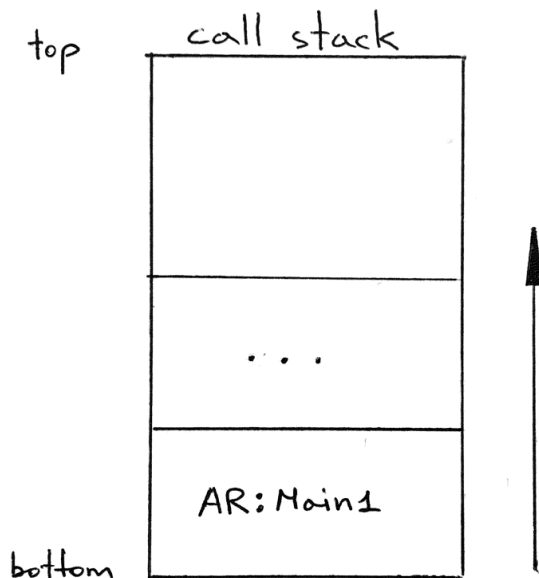
什么是堆栈？一组是基于“数据结构，后进先出”的政策（LIFO），这意味着添加到堆栈的最新产品所散发出来的第一个。这就像一组盘子，你把一个盘子放在（“推”）盘子堆的顶部，如果你需要拿一个盘子，你从盘子堆的顶部取下一个（你“弹出”盘子））：



我们的堆栈实现将具有以下方法：

- push（将项目推入堆栈）
- 弹出（从堆栈中弹出一个项目）
- peek（返回堆栈顶部的项目而不删除它）

按照我们的惯例，我们的堆栈将向上增长：



我们将如何在代码中实现堆栈？一个非常基本的实现可能如下所示：

```

类 堆栈:
    def __init__ (自我):
        自我.项目 = []

    def push ( self , item ):
        self . 项目.追加 (项目)

    def pop ( self ):
        返回 self . 项目.流行()

    def peek ( self ):
        返回 self . 项目[ - 1 ]

```

这几乎就是我们的调用堆栈实现的样子。我们将更改一些变量名称以反映调用堆栈将存储激活记录并添加一个 `__str__()` 方法来打印堆栈内容的事实：

```

类 CallStack :
    def __init__ ( self ):
        self ._记录 = []

    def push ( self , ar ):
        self ._记录.附加( ar )

    def pop ( self ):
        返回 self ._记录.流行()

    def peek ( self ):
        返回 self ._records [ - 1 ]

    def __str__ ( self ):
        s = ' \n ' 。加入 (再版 (AR ) 为 AR 在 扭转 (自._records ) )
        小号 = 'F 'CALL堆栈\ n {S} \ n '
        返回 小号

    def __repr__ ( self ):
        返回 self .___str__ ()

```

`__str__()`方法通过以相反的顺序迭代激活记录并连接每个记录的字符串表示来生成最终结果，从而生成调用堆栈内容的字符串表示。`__str__()`方法以相反的顺序打印内容，以便标准输出显示我们的堆栈正在增长。

现在，什么是**激活记录**？对于我们的目的，活动记录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，过程调用的活动记录将包含其形式参数及其局部变量的当前值。

让我们来看看我们将如何在代码中表示激活记录：

```

类 ARType (枚举):
    PROGRAM = 'PROGRAM'

类 ActivationRecord :
    def __init__ ( self , name , type , nesting_level ):
        self .姓名 = 姓名
        自我.类型 = 类型
        self .nesting_level = nesting_level
        self .成员 = {}

    def __setitem__ ( self , key , value ):
        self .成员[键] = 值

    def __getitem__ ( self , key ):
        返回 self .成员[键]

    def get ( self , key ):
        返回 self .成员.得到 (键)

    def __str__ ( self ):
        lines = [
            '{level}: {type} {name}' 。格式 (
                级别=自我.nesting_level ,
                类型=自我.类型.值,
                名称=自我.名,
            )
        ]
        为 名称, VAL 在 自我.成员.项目 ( ):
            行.追加( f ' {name:<20}: {val}' )

        s = ' \n ' 。加入 (行)
        返回 s

    def __repr__ ( self ):
        返回 self .___str__ ()

```

有几点值得一提：

一种。该ActivationRecord类的构造函数有三个参数：

- 激活记录的名称（简称AR）；我们将使用程序名称以及过程/函数名称作为相应AR的名称
- 激活记录的类型（例如，PROGRAM）；这些在称为ARType（激活记录类型）的单独枚举类中定义
- 活动记录的nesting_level；AR的嵌套级别对应于相应过程或函数声明的范围级别加一；程序的嵌套级别将始终设置为 1，您很快就会看到

湾的成员字典表示将被用于保持约的程序的特定调用信息的存储器。我们将在下一篇文章中更详细地介绍这一点

C. 所述ActivationRecord类实现特殊__setitem__ () 和__getitem__ () 方法，得到激活记录对象类似字典的接口，用于存储键-值对，并用于通过按键访问值：AR ["X"] = 7和AR ["x"] 的

d. 该GET () 方法是另一种方式来获得由键的值，但不是抛出一个异常，该方法将返回无，如果不存在，关键在成员字典呢。

e. 所述__str__ () 方法返回的激活记录的内容的字符串表示

让我们使用 Python shell 查看调用堆栈和激活记录：

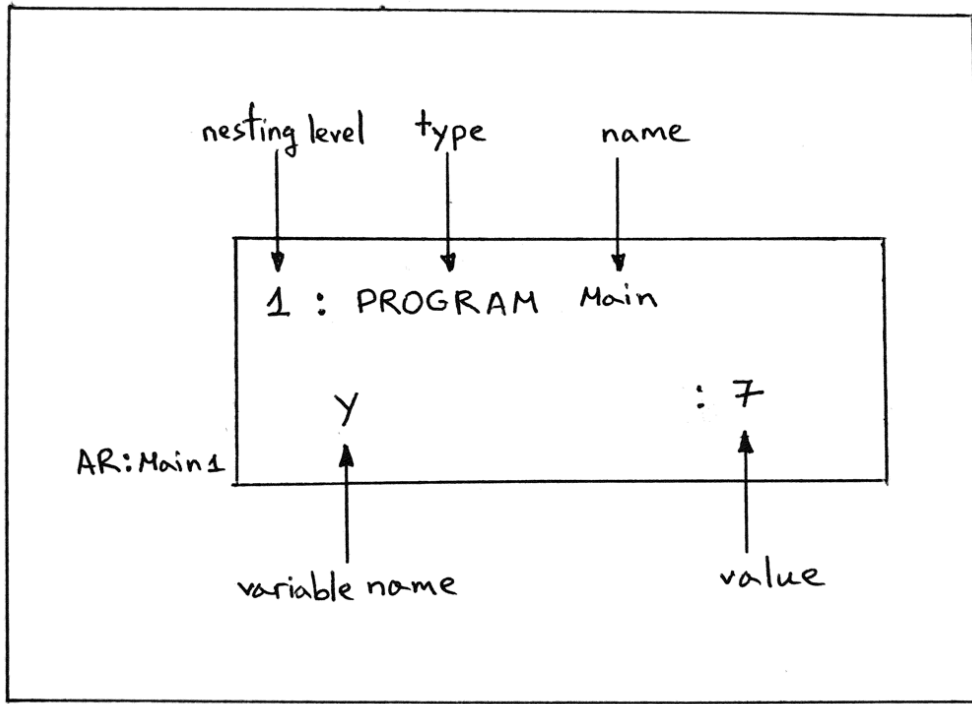
```
>>> from spi import CallStack , ActivationRecord , ARType
>>> stack = CallStack ()
>>> stack
CALL STACK

>>> ar = ActivationRecord ( name = 'Main' , type = ARType . PROGRAM , nesting_level = 1 )
>>>
>>> ar
1 : PROGRAM Main
>>>
>>> ar [ 'y' ] = 7
>>>
>>> ar
1 : PROGRAM Main
   y                : 7
>>>
>>> stack
CALL STACK

>>> 堆栈。push ( ar )
>>>
>>> stack
CALL STACK
1 : PROGRAM Main
   y                : 7

>>>
```

在下图中，您可以从上面的交互会话中看到激活记录的内容描述：



AR :Main1表示在嵌套级别1 中名为Main的程序的激活记录。

现在我们已经介绍了新的内存系统，让我们回答以下问题。

4.为什么我们要用调用堆栈替换 GLOBAL_MEMORY 字典？

原因是为了简化我们的实现并统一访问在程序级别定义的全局变量以及过程和函数参数及其局部变量。

在下一篇文章中，我们将看到它们是如何组合在一起的，但现在让我们来看看Interpreter类的变化，我们将前面描述的调用堆栈和激活记录充分利用起来。

以下是我们今天要做的所有解释器更改：

- 1.用调用栈替换GLOBAL_MEMORY字典
2. 更新visit_Program方法，使用调用栈来压入和弹出一个将保存全局变量值的激活记录
- 3.更新visit_Assign方法，在调用栈顶部的激活记录中存储一个键值对
4. 更新visit_Var方法以从调用堆栈顶部的活动记录中通过其名称访问值
- 5.添加log方法，更新visit_Program方法，在解释程序时使用它打印调用栈的内容

让我们开始吧，好吗？

1. 首先，让我们用我们的调用堆栈实现替换GLOBAL_MEMORY字典。我们需要做的就是更改Interpreter构造函数：

```
class Interpreter ( NodeVisitor ):
    def __init__ ( self , tree ):
        self . 树 = 树
        自我。GLOBAL_MEMORY = {}
```

对此：

```
class Interpreter ( NodeVisitor ):
    def __init__ ( self , tree ):
        self . 树 = 树
        自我。call_stack = CallStack ()
```

2. 现在，让我们更新visit_Program 方法：

旧代码：

```
def visit_Program ( self , node ):
    self 。访问 ( 节点。块 )
```

新代码：

```
def visit_Program ( self , node ):
    program_name = node 。姓名

    AR = ActivationRecord (
        名称=程序名,
        类型= ARType 。PROGRAM ,
        nesting_level = 1 ,
    )
    自我。调用堆栈。推( ar )

    自我。访问 ( 节点。块 )

    自我。调用堆栈。流行()
```

让我们解开上面更新的方法中发生的事情：

- 首先，我们创建一个活动记录，为其指定程序名称、程序类型和嵌套级别 1
- 然后我们将激活记录压入调用栈；我们在做任何其他事情之前这样做，以便解释器的其余部分可以使用调用堆栈和堆栈顶部的单个活动记录来存储和访问全局变量
- 然后我们像往常一样评估程序的主体。同样，当我们的解释器评估程序体时，它使用调用堆栈顶部的激活记录来存储和访问全局变量
- 接下来，就在退出visit_Program方法之前，我们从调用堆栈中弹出激活记录；我们不再需要它了，因为此时解释器对程序的执行已经结束，我们可以安全地丢弃不再使用的活动记录

3. 接下来，让我们更新visit_Assign方法以在调用堆栈顶部的激活记录中存储键值对：

旧代码：

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

新代码：

```
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
```

在上面的代码中，我们使用peek()方法获取堆栈顶部的激活记录（通过visit_Program方法压入堆栈的那个），然后使用该记录存储值var_value使用var_name作为钥匙。

4. 接下来，让我们更新visit_Var方法以从调用堆栈顶部的活动记录中按名称访问值：

旧代码：

```
def visit_Var ( self , node ):
    var_name = node 。值
    var_value = self 。全局内存。get ( var_name )
    返回 var_value
```

新代码：

```
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
```

再次如您所见，我们使用peek()方法获取顶部（也是唯一的）激活记录 - 被visit_Program方法压入堆栈以保存所有全局变量及其值的激活记录 - 然后获取一个与var_name 键关联的值。

5.我们将在Interpreter类中进行的最后一个更改是添加一个log方法，并在解释器评估程序时使用log方法打印调用堆栈的内容：

```
def log ( self , msg ):
    if _SHOULD_LOG_STACK :
        print ( msg )

def visit_Program ( self , node ):
    program_name = node 。 名称
    自我。日志( f 'ENTER: PROGRAM {program_name}' )

    AR = ActivationRecord (
        名称=程序名,
        类型= ARType 。 PROGRAM ,
        nesting_level = 1 ,
    )
    自我。调用堆栈。推( ar )

    自我。日志( str ( self . call_stack ))

    自我。访问 ( 节点。块 )

    自我。log ( f 'LEAVE: PROGRAM {program_name}' )
    self 。 日志( str ( self . call_stack ))

    自我。调用堆栈。流行()
```

仅当全局变量 _SHOULD_LOG_STACK 设置为 true 时才会记录消息。变量的值将由 “--stack” 命令行选项控制。首先，让我们更新主函数并添加 “--stack” 命令行选项来打开和关闭调用堆栈内容的记录：

```
def main ():
    parser = argparse 。 ArgumentParser (
        description = 'SPI - Simple Pascal Interpreter'
    )
    解析器.add_argument ( 'inputfile' , help = 'Pascal source file' )
    解析器.add_argument (
        '--scope' ,
        help = 'Print scope information' ,
        action = 'store_true' ,
    )
    解析器.add_argument (
        '--stack' ,
        help = '打印调用堆栈' ,
        action = 'store_true' ,
    )
    args = parser 。 解析参数()

    全局 _SHOULD_LOG_SCOPE , _SHOULD_LOG_STACK

    _SHOULD_LOG_SCOPE , _SHOULD_LOG_STACK = args 。 范围, 参数。堆
```


现在，让我们测试一下我们更新的解释器。从GitHub

(<https://github.com/rspivak/lbasi/tree/master/part17>)下载解释器并使用-h命令行选项运行它以查看可用的命令行选项：

```
$ python spi.py -h
用法: spi.py [ -h ] [ --scope ] [ --stack ] inputfile中

SPI - 简单的帕斯卡解释器

位置参数:
  输入文件Pascal源文件

可选参数:
  -h, --help 显示此帮助信息并退出
  --scope 打印范围信息
  --stack 打印调用堆栈
```

从GitHub (<https://github.com/rspivak/lbasi/tree/master/part17>)下载以下示例程序或保存到文件part17.pas

```
程序 主程序;
var x , y : 整数;
开始 { 主要 }
  y := 7 ;
  x := ( y + 3 ) * 3 ;
结束。 { 主要的 }
```

使用part17.pas文件作为其输入文件和“--stack”命令行选项运行解释器，以查看解释器执行源程序时调用堆栈的内容：

```
$ python spi.py part17.pas --stack
输入: 程序主
调用堆栈
1 : 程序主

离开: 程序主要
调用堆栈
1 : 程序主
  y : 7
  x : 30
```

任务完成！我们实现了一个新的内存系统，可以支持程序、过程调用和函数调用。我们已经用基于调用堆栈和激活记录的新系统替了解释器当前的内存系统，由GLOBAL_MEMORY字典表示。

这就是今天的全部内容。在下一篇文章中，我们将扩展解释器以使用调用堆栈和激活记录执行过程调用。这对我们来说将是一个巨大的里程碑。所以请继续关注我们下期再见！

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=5d5ca8c07bff5452ea443d8319e7703d)

- 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)
4. 用故事引导 (https://www.amazon.com/gp/product/0814420303/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0814420303&linkCode=as2&tag=russblo0b-20&linkId=bee8bb0ac4fa2fb1ce587e093b6cfe6c)
5. 一个[维基百科](https://en.wikipedia.org/wiki/John_Stephen_Akhwari)的文章 (https://en.wikipedia.org/wiki/John_Stephen_Akhwari)在约翰·斯蒂芬·阿卡瓦里

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 10.

5 years ago • 22 comments

Today we will continue closing the gap between where we are right now ...

Let's Build A Web Server. Part 2.

6 years ago • 61 comments

Remember, in Part 1 I asked you a question: "How do you run a Django application, ...

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 ...

Let's B Server.

6 years a

"We lear have to i Part 2 yc

[8 Comments](#) [Ruslan's Blog](#) [Disqus' Privacy Policy](#)[Login](#)[Recommend](#) 3 [Tweet](#) [Share](#)[Sort by Best](#)

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

**LanaOblivion** • 2 years ago

Is there or will be a part 18?

1 ^ | v • Reply • Share ›

**rspivak** Mod → LanaOblivion • 2 years ago

Yes, the next part is coming out soon.

1 ^ | v • Reply • Share ›

**GilDev** • 2 years ago

Always great articles! Would love to see a "Let's Build A Simple Compiler" sequel!

1 ^ | v • Reply • Share ›

**Davide Casale** • 2 years ago

Hello sir. I am a 17 years old developer, I started programming when I was 13 and I think I achieved a lot in this four years about programming and developing so I decided to challenge my-self trying to create my own interpreter for my personal language. I decided to create it using C++ (I thought that a compiled language is better to create an interpreter), I read some books about the argument and I took a look at your guide, too, and I love it! The project is "Work In Progress" but I would really appreciate if you can give me a little feedback about what I am actually doing, I am a self-taught and it is hard for me to have someone evaluate my work :)

Project Link: <https://github.com/Davi0k/EMERALD>

Thanks in advance, have a great day!

1 ^ | v • Reply • Share ›

**solstice333** • a year ago • edited

@**rspivak** thanks for putting this out. This has been a great way to dive into compilers by experience. Here's my take on everything so far if anyone is interested: <https://github.com/solstice333/Compiler>. I tried to leverage a few other things and ended up refactoring a lot as I advanced through your series.

^ | v • Reply • Share ›

**Guga Loks** • 2 years ago

Please, next part can you add some concepts like Arrays and OO?

^ | v • Reply • Share ›

**hotel** • 2 years ago

thanks for the series! one question though: is there a reason you don't build the symbol table(s) in a first pass and then do type analysis in a second pass? does Pascal (or the Pascal-like source language here) only allow backward references?

^ | v • Reply • Share ›

**James** • 2 years ago


I finished it, thanks

<https://github.com/shiftone/Compiler>

^ | v • Reply • Share ›

 [github \(https://github.com/rspivak/\)](https://github.com/rspivak/)

 [推特 \(https://twitter.com/rspivak\)](https://twitter.com/rspivak)

 [链接 \(https://linkedin.com/in/ruslanspivak/\)](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/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。