让我们构建一个简单的解释器。第 15 部分。 (https://ruslanspivak.com/lsbasi-part15/)

日期 2019 年 6 月 21 日星期五

> “我走得很慢，但我从不后退。” - 亚伯拉罕·林肯

我们又回到了定期安排的节目中！:)

在继续讨论识别和解释过程调用的主题之前，让我们进行一些更改以稍微改进我们的错误报告。到目前为止，如果从文本中获取新标记、解析源代码或进行语义分析时出现问题，堆栈跟踪将直接抛出一个非常通用的消息。我们可以做得更好。

为了提供更好的错误消息，指出代码中出现问题的位置，我们需要向解释器添加一些功能。让我们这样做，并在此过程中进行一些其他更改。这将使口译员更加用户友好，并让我们有机会在系列赛的“短暂”休息后锻炼我们的肌肉。它还让我们有机会为我们将在以后的文章中添加的新功能做好准备。

今天的目标：

- 改进词法分析器、解析器和语义分析器中的错误报告。相反，与像非常普通的消息堆栈跟踪“无效语法”，我们希望看到更多的类似的东西有用“的 SyntaxError：意外的标记- >令牌（TokenType。SEMI ‘;’ ，位置= 23：13）”
- 添加“-scope”命令行选项以打开/关闭范围输出
- 切换到 Python 3。从现在开始，所有代码将只在 Python 3.7+ 上测试

让我们通过首先更改我们的词法分析器来破解并开始展示我们的编码肌肉。

以下是我们今天要在词法分析器中进行的更改列表：

1. 我们将添加错误代码和自定义异常：LexerError，ParserError和SemanticError
2. 我们将向Lexer类添加新成员以帮助跟踪标记的位置：lineno和column
3. 我们将修改提前方法来更新词法分析器的LINENO和列 变量
4. 我们将更新错误方法以引发LexerError异常，其中包含有关当前行和列的信息

5. 我们将在TokenType枚举类中定义令牌类型（Python 3.4 中添加了对枚举的支持）
6. 我们将添加代码以从TokenType枚举成员中自动创建保留关键字
7. 我们将在Token类中添加新成员：lineno和column以跟踪标记的行号和列号，相应地，在文本中
8. 我们将重构get_next_token方法代码以使其更短，并拥有处理单字符标记的通用代码

1. 我们先定义一些错误代码。我们的解析器和语义分析器将使用这些代码。我们还定义了以下错误类：LexerError，ParserError和SemanticError词法，句法，并且相应地，语义错误：

```
从 枚举 导入 枚举

class  ErrorCode ( Enum ):
    UNEXPECTED_TOKEN  =  'Unexpected token'
    ID_NOT_FOUND      =  'Identifier not found'
    DUPLICATE_ID      =  'Duplicate id found'

class  Error ( Exception ):
    def  __init__ ( self ,  error_code = None ,  token = None ,  message = None ):
        self 。error_code  =  error_code
        self 。token  =  token
        # 在消息
        self前添加异常类名。message  =  f '{self.__class__.__name__}: {message}'

类 LexerError （错误）：
    通过

类 解析器错误（错误）：
    通过

类 语义错误（错误）：
    通过
```

ErrorCode是一个枚举类，其中每个成员都有一个名称和一个值：

```
>>>  from  enum  import  Enum
>>>
>>>  class  ErrorCode ( Enum ):
...      UNEXPECTED_TOKEN  =  'Unexpected token'
...      ID_NOT_FOUND      =  'Identifier not found'
...      DUPLICATE_ID      =  'Duplicate id found'
.. .
>>>  ErrorCode
< enum  'ErrorCode' >
>>>
>>>  ErrorCode 。ID_NOT_FOUND
<错误代码。ID_NOT_FOUND :  '未找到标识符'
```

该错误的基类的构造函数有三个参数：

- error_code : ErrorCode.ID_NOT_FOUND 等
- token : Token 类的一个实例
- message：包含有关问题的更详细信息的消息

正如我之前提到的，LexerError用来表示在词法分析器中遇到错误，ParserError是在分析阶段相关的语法错误，SemanticError是语义错误。

2. 为了提供更好的错误信息，我们希望在源文本中显示问题发生的位置。为了能够做到这一点，我们需要在生成标记时开始跟踪词法分析器中的当前行号和列。让我们添加LINENO和列字段的词法分析器 类：

```
class  Lexer ( object ):
    def  __init__ ( self ,  text ):
        ...
        # self.pos 是 self.text
        self的索引。pos  =  0
        自我。current_char  =  self 。文本[自我。pos ]
        # 标记行号和列号
        self . lineno  =  1
        self 。列 =  1
```

3. 下一个我们需要做的改变是在advance方法中遇到新行时重置lineno和column，并且在self.pos 指针每次advance时增加column值：

```
DEF 提前（自）：
    “”“提前`pos`指针，并设置`current_char`变量”“”
    如果 自我。current_char  ==  ' \n ' :
        self 。lineno  +=  1
        self . 列 =  0

    自我。pos  +=  1
    如果 self 。pos  >  len ( self . text )  -  1 :
        self . current_char  =  None   # 表示输入结束
    else :
        self . current_char  =  self 。文本[自我。pos ]
        自我。列 +=  1
```

有了这些更改，每次我们创建一个标记时，我们都会将当前的lineno和column从词法分析器传递给新创建的标记。

4. 让我们更新error方法以抛出一个LexerError异常，并带有更详细的错误消息，告诉我们词法分析器阻塞的当前字符及其在文本中的位置。

```
def  error ( self ):
    s  =  "'{lexeme}' 行上的词法分析器错误：{lineno} 列：{column}" 。格式（
        词位=自我。current_char ，
        LINENO =自我。LINENO ，
        柱=自我。柱，
    ）
    加注 LexerError （消息=小号）
```

5. 不是将令牌类型定义为模块级变量，而是将它们移动到称为TokenType的专用枚举类中。这将帮助我们简化某些操作并使我们的代码的某些部分更短一些。

老款式：

```
# 令牌类型
PLUS   =  'PLUS'
MINUS  =  'MINUS'
MUL    =  'MUL'
...
```

新风格：

```
class  TokenType ( Enum ):
    # 单字符标记类型
    PLUS           =  '+'
    MINUS          =  '-'
    MUL            =  '*'
    FLOAT_DIV      =  '/'
    LPAREN         =  '('
    RPAREN         =  ')'
    SEMI           =  ';'
    点           =  '.'
    COLON          =  ':'
    COMMA          =  ','
    # 保留字块
    PROGRAM        =  'PROGRAM'    # 标记块的开始
    INTEGER        =  'INTEGER'

    INTEGER_DIV    =  'DIV'
    VAR            =  'VAR'
    PROCEDURE      =  'PROCEDURE'
    BEGIN          =  'BEGIN'
    END            =  'END'        # 标记块的结束
    # misc
    ID             =  'ID'
    INTEGER_CONST  =  'INTEGER_CONST'
    REAL_CONST     =  'REAL_CONST'
    ASSIGN         =  ':='
    EOF            =  'EOF'
```

6. 我们过去常常在必须添加新的标记类型时手动将项目添加到RESERVED_KEYWORDS字典中，该类型也是保留关键字。如果我们想添加一个新的STRING令牌类型，我们必须

- (a) 创建一个新的模块级变量STRING = ' STRING '
- (b) 手动将其添加到RESERVED_KEYWORDS 字典

现在我们有了TokenType枚举类，我们可以删除上面的手动步骤**(b)**并将标记类型仅保留在一个地方。这是“两个太多 (https://www.codesimplicity.com/post/two-is-too-many/)”规则在起作用 - 继续前进，添加新关键字标记类型所需的唯一更改是将关键字放在TokenType枚举类中的PROGRAM和END之间，并且_build_reserved_keywords函数将照顾其余的：

```
def  _build_reserved_keywords ():
    """构建保留关键字字典。

    该函数依赖于这样一个事实：在 TokenType
    枚举中，保留关键字块的开头
    用 PROGRAM 标记，块的结尾用
    END 关键字标记。


在定义顺序tt_list = list ( TokenType ) start_index = tt_list 。索引( TokenType . PROGRA









    end_index  =  tt_list 。指数（TokenType 。END ）
    reserved_keywords  =  {
        token_type 。value :  token_type
        for  token_type  in  tt_list [ start_index : end_index  +  1 ]
    }
    return  reserved_keywords


RESERVED_KEYWORDS  =  _build_reserved_keywords ()
```

从函数的文档字符串中可以看出，该函数依赖于这样一个事实，即TokenType枚举中的保留关键字块由PROGRAM和END 关键字标记。

该函数首先将TokenType转成列表（保留定义顺序），然后获取块的起始索引（以PROGRAM关键字标记）和块的结束索引（以END关键字标记）。接下来，它使用字典理解来构建一个字典，其中键是枚举成员的字符串值，值是TokenType成员本身。

```
>>> from spi import _build_reserved_keywords
>>> from pprint import pprint
>>> pprint ( _build_reserved_keywords ())  # 'pprint' 对键进行排序
{ 'BEGIN' : < TokenType . BEGIN : 'BEGIN' > ,
 'DIV' : < TokenType 。INTEGER_DIV : 'DIV' > ,
 'END' : < TokenType 。结束： '结束'
 : <令牌类型。整数： '整数' > ,
 '过程' : < TokenType 。程序： '程序' > ,
 '程序' : <令牌类型。程序： '程序' > ,
 '真实' : < TokenType 。REAL : 'REAL' > ,
 'VAR' : < TokenType 。VAR : 'VAR' > }
```

7. 下一个变化是在Token类中添加新成员，即lineno和column，以跟踪文本中标记的行号和列号

```
class Token ( object ):
    def __init__ ( self , type , value , lineno = None , column = None ):
        self 。类型 = 类型
        self 。价值 = 价值
        自我。lineno = lineno
        self . 列 = 列

    def __str__ ( self ):
        """类实例的字符串表示。

        示例：
            >>> Token(TokenType.INTEGER, 7, lineno=5, column=10)
            Token(TokenType.INTEGER, 7, position=5:10)
        """
        return 'Token({type}, {value},位置= {LINENO}：{柱}）”。格式（
            类型=自我。类型，
            值=再版（自我。值），
            LINENO =自我。LINENO ，
            柱=自我。柱，
        ）

    def __repr__ ( self ):
        返回 self 。__str__ ()
```

8. 现在，get_next_token方法更改。感谢枚举，我们可以通过编写生成单字符标记的通用代码来减少处理单字符标记的代码量，并且在我们添加新的单字符标记类型时不需要更改：

而不是像这样的大量代码块：

```
如果 自. current_char  ==  ';' :
    自己。提前()
    返回 令牌( SEMI ,  ';' )

如果 自. current_char  ==  ':' :
    self 。提前()
    返回 令牌( COLON ,  ':' )

如果 自. current_char  ==  ',' :
    self 。提前（）
    返回 令牌（逗号， ', ' ）
...
```

我们现在可以使用这个通用代码来处理所有当前和未来的单字符标记

```
# 单字符标记
try :
    # 按值获取枚举成员，例如
    # TokenType(';') --> TokenType.SEMI
    token_type  =  TokenType ( self . current_char )
除了 ValueError :
    # 没有值等于 self.current_char 的枚举成员
    自我。error ()
else :
    # 创建一个以单字符词位作为值的
    标记 token  =  Token (
        type = token_type ,
        value = token_type . value ,   # eg ';', '.', etc
        lineno =自我。lineno ，
        column = self 。列,
    )
    自我。提前（）
    返回 令牌
```

可以说它不如一堆if块可读，但是一旦你理解了这里发生了什么，它就非常简单了。Python 枚举允许我们通过值访问枚举成员，这就是我们在上面的代码中使用的。它是这样工作的：

- 首先，我们试图得到一个TokenType由值的成员self.current_char
- 如果操作抛出ValueError异常，则表示我们不支持该令牌类型
- 否则，我们会使用相应的令牌类型和值创建正确的令牌。

此代码块将处理所有当前和新的单字符标记。为了支持新的令牌类型，我们需要做的就是将新的令牌类型添加到TokenType定义中，就是这样。上面的代码将保持不变。

在我看来，这个通用代码是一个双赢的局面：我们学到了更多关于 Python 枚举的知识，特别是如何通过值访问枚举成员；我们编写了一些通用代码来处理所有单字符标记，作为副作用，我们减少了处理这些单字符标记的重复代码量。

下一站是解析器更改。

以下是我们今天将在解析器中进行的更改列表：

1. 我们将更新解析器的错误方法以抛出带有错误代码和当前标记的ParserError异常
2. 我们将更新eat方法调用修改后的error 方法
3. 我们将重构声明方法并将解析过程声明的代码移动到一个单独的方法中。

1. 让我们更新解析器的错误方法以抛出包含一些有用信息的ParserError异常

```
def  error ( self ,  error_code ,  token ):
    raise  ParserError (
        error_code = error_code ,
        token = token ,
        message = f '{error_code.value} -> {token}' ,
    )
```

2.现在让我们修改eat方法来调用更新的错误 方法

```
def  eat ( self ,  token_type ):
    # 比较当前标记类型与传递的标记
    # 类型，如果它们匹配，则“吃”当前标记
    # 并将下一个标记分配给 self.current_token，
    # 否则引发异常。
    如果 自. current_token 。type  ==  token_type :
        self 。current_token  =  self 。get_next_token ()
    其他：
        自我。错误(
            error_code = ErrorCode . UNEXPECTED_TOKEN,
            令牌=自我。current_token ,
        )
```

3. 接下来，让我们更新声明的文档字符串并将解析过程声明的代码移动到单独的方法 procedure_declaration 中：

```
def 声明( self ):
    """
    声明 : (VAR (variable_declaration SEMI)+)? procedure_declaration*
    """
    声明 =  []

    如果 自. current_token 。类型 == 令牌类型。VAR :
        自我。吃（TokenType 。VAR ）
        ，而 自我。current_token 。类型 == 令牌类型。ID :
            var_decl  =  self 。variable_declaration ()
            声明。扩展（var_decl ）
            自我。吃( TokenType . SEMI )

    而 自我。current_token 。类型 == 令牌类型。程序:
        proc_decl  =  self 。procedure_declaration ()
        声明。追加（proc_decl ）

    返回 声明

def  procedure_declaration ( self ):
    """procedure_declaration :
         PROCEDURE ID (LPARENformal_parameter_list RPAREN)? SEMI block SEMI
    """
    self 。吃（TokenType 。PROCEDURE ）
    proc_name中 = 自我。current_token 。重视
    自我。吃( TokenType . ID )
    params  =  []

    如果 自. current_token 。类型 == 令牌类型。LPAREN :
        自我。吃（TokenType 。LPAREN ）
        PARAMS  = 自我。formal_parameter_list ()
        自我。吃（TokenType . RPAREN ）

    自我。吃（TokenType 。SEMI ）
    block_node  = 自我。block ()
    proc_decl  =  ProcedureDecl ( proc_name ,  params ,  block_node )
    self . 吃( TokenType . SEMI )
    返回 proc_decl
```

这些都是解析器的变化。现在，我们将进入语义分析器。

最后，这是我们将在语义分析器中进行的更改列表：

1. 我们将向SemanticAnalyzer类添加一个新的错误方法，以抛出带有一些附加信息的SemanticError异常
2. 我们将更新visit_VarDecl以通过调用具有相关错误代码和令牌的错误方法来发出错误信号
3. 我们还将更新visit_Var以通过调用具有相关错误代码和令牌的错误方法来发出错误信号

4. 我们将为ScopedSymbolTable和SemanticAnalyzer添加一个log方法，并将所有打印语句替换为对相应类中的self.log 的调用
5. 我们将添加一个命令行选项“--scope”来打开和关闭范围日志记录（默认情况下它将关闭）以控制我们希望解释器的“嘈杂”程度
6. 我们将添加空的visit_Num和visit_UnaryOp 方法

1. 第一件事。让我们添加error方法来抛出一个SemanticError异常，并带有相应的错误代码、标记和消息：

```
def  error ( self ,  error_code ,  token ):
    raise  SemanticError (
        error_code = error_code ,
        token = token ,
        message = f '{error_code.value} -> {token}' ,
    )
```

2. 接下来，让我们更新visit_VarDecl以通过调用带有相关错误代码和令牌的error方法来发出错误信号

```
def  visit_VarDecl ( self ,  node ):
    type_name  =  node 。类型节点。值
    type_symbol  =  self 。current_scope 。查找（type_name ）

    # 我们拥有创建变量符号所需的所有信息。
    # 创建符号并将其插入到符号表中。
    var_name  = 节点。变量节点。值
    var_symbol  =  VarSymbol ( var_name ,  type_symbol )

    # 如果表已经有一个
    同名
    的符号，则发出错误信号# if  self 。current_scope 。查找（var_name ，  current_scope_on
        self 。错误（
            error_code = ErrorCode . DUPLICATE_ID ，
            token = node . var_node . token ，
        ）

    自我。current_scope 。插入（var_symbol ）
```

3.我们还需要更新visit_Var方法通过调用发出错误信号错误有相关的错误代码和标记方法

```
def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.current_scope.lookup(var_name)
    if var_symbol is None:
        self.error(error_code=ErrorCode.ID_NOT_FOUND, token=node.token)
```

现在语义错误会报如下：

```
语义错误：发现重复的 id -> Token(TokenType.ID, 'a', position=21:4)
```

或者

```
语义错误：未找到标识符 -> Token(TokenType.ID, 'b', position=22:9)
```

4. 让我们将log方法添加到ScopedSymbolTable和SemanticAnalyzer，并将所有打印语句替换为对self.log 的调用：

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

如您所见，仅当全局变量 _SHOULD_LOG_SCOPE 设置为 true 时才会打印该消息。我们将在下一步中添加的—scope命令行选项将控制 _SHOULD_LOG_SCOPE 变量的值。

5. 现在，让我们更新主函数并添加一个命令行选项“--scope”来打开和关闭范围日志（默认情况下是关闭的）

```
解析器 = 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',
)
args = parser.parse_args()
全局 _SHOULD_LOG_SCOPE
_SHOULD_LOG_SCOPE = 参数.范围
```

这是打开开关的示例：

```
$ python spi.py idnotfound.pas --scope
输入范围：全局
插入：整数
插入：真实
查找：整数。（范围名称：全局）
查找：（范围名称：全局）
插入：一个
查找： B. （范围名称：全局）
语义错误：未找到标识符 -> Token(TokenType.ID, 'b', position=6:9)
```

并且范围注销（默认）：

```
$ python spi.py idnotfound.pas
语义错误：未找到标识符 -> Token(TokenType.ID, 'b', position=6:9)
```

6.添加空visit_Num和visit_UnaryOp 方法

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

def  visit_UnaryOp ( self ,  node ):
    通过
```

这些是目前对我们的语义分析器的所有更改。

请参阅GitHub (https://github.com/rspivak/lsbasi/tree/master/part15/)以获取具有不同错误的 Pascal 文件以尝试更新的解释器并查看解释器生成的错误消息。

这就是今天的全部内容。您可以在GitHub (https://github.com/rspivak/lsbasi/tree/master/part15/)上找到 (https://github.com/rspivak/lsbasi/tree/master/part15/)今天文章解释器的完整源代码。在下一篇文章中，我们将讨论如何识别（即如何解析）过程调用。敬请期待，我们下期再见！

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 9.

5 years ago • 42 comments

I remember when I was in university (a long time ago) and learning systems …

Let's Build A Simple Interpreter. Part 6.

6 years ago • 20 comments

Today is the day :) "Why?" you might ask. The reason is that today we're …

Let's Build A Simple Interpreter. Part 12.

5 years ago • 29 comments

"Be not afraid of going slowly; be afraid only of standing still." - Chinese …

Let's B Server.

6 years a

"We lear have to i Part 2 yo

1 • Reply • Share ›

Shreevatsa R • 2 years ago

Hooray! Thanks for resuming this; have loved this series so far. Good luck with future episodes :-)

1 • Reply • Share ›

rspivak Mod → Shreevatsa R • 2 years ago

Thank you! :)

• Reply • Share ›

Hermes Passer • 2 years ago

welcome back

1 • Reply • Share ›

rspivak Mod → Hermes Passer • 2 years ago

Thanks :)

• Reply • Share ›

Immad • 2 years ago

I waited for part 15 for 2 years. haha. Keep the good work up.

1 • Reply • Share ›

rspivak Mod → Immad • 2 years ago

Thank you

• Reply • Share ›

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