MiniScript： 编译原理大程报告

简介

MiniScript是一个简单，灵活，强大的语言。它的主要灵感来源于JavaScript，也吸收了Python，Lua等语言的做法。

我们交上来的文件里面，主程序是miniscript.py，你可以用一个命令行参数去运行这个程序，参数就是文件名，然后我们的程序可以运行出结果。你必须先安装Python3（要求3.5以上），然后安装PLY这个Python库。

mslexer.py文件是做词法分析的。msparser.py是做语法分析，语法制导翻译和语法生成的。vm.py是虚拟机，value.py, manager.py和其他文件是辅助文件。

测试用例在example目录下，其中hello.ms做了一个hello world，fibo.ms做了一个斐波那契，recursive_fibo.ms做了一个递归的斐波那契，object_fibo.ms做了一个面向对象的斐波那契。prototype.ms展示了如何使用原型继承，student.ms做了一个Student类和它的两个实例，展示了如何利用原型继承来模拟类-对象的继承结构。

郑雅心和吴佳佳负责了词法分析和语法分析，邹遥负责了语法制导翻译，代码生成和虚拟机。

简单

它简单，因为整个语言的类型系统非常小巧：MiniScript中只有五种类型：int, real, str, function和table。没有array或者list，尽管你可以写`[1, 2, "hello"]`这样的表达式。因为你写的`[1, 2, "hello"]`会被编译器转换成`{0: 1, 1:2, 2: "hello"}`。也就是说，数组其实也就是一个下标到元素的映射而已。这样的思想借鉴自Lua。MiniScript中也没有结构体或者类，因为你可以用创建table一样的方法来创建一个对象：

```
student = {
    "age": 21,
    "changeAge": function(self: table, age: int){
        self.age = age
        return 0
    }
}
student.changeAge(student, 22)
print student.age
return 0
```

点运算符看上去神奇，其实只不过是下标运算符的一种伪装而已。上面的代码和下面的代码其实是等价的

```
student = {
    "age": 21,
    "changeAge": function(self: table, age: int){
        self["age"] = age
        return 0
    }
}
student["changeAge"](student, 22)
print student["age"]
return 0
```

然而引入点运算符，让我们的语言看上去有了面向对象的特征。在C++或者Java这样的语言里面，数组（或者高级一点的`vector`，`ArrayList`），`table`（也就是标准库里面的`map`）以及对象（由class派生出来的实例）是截然不同的。然而MiniScript吸收了JavaScript和Lua的经验，将它们统一为一种数据类型，极大地降低了语言的复杂度。

灵活

MiniScript灵活，是因为它提供了非常宽松的语法，让你随心所欲地编程。在MiniScript里面，函数是一种"一等公民"，这意味着它们可以被赋值给一个变量，可以作为函数参数传递给另一个函数，也可以作为函数返回值返回。

同时，MiniScript提供了基于函数的作用域。在MiniScript里面，作用域是以函数为单位而不是以块（比如for循环）为单位的。也就是说，在for循环的后面你可以访问for循环里面定义的变量。

在MiniScript里面，函数可以嵌套定义，定义在某函数f里面的函数g可以访问（甚至修改）函数f里面的本地变量。但是，请注意，如果你的内部函数g访问了外部函数f里面的本地变量，请不要将g从f中返回。因为MiniScript不支持JavaScript风格的闭包（闭包的实现太复杂了）。返回一个访问了本地变量的函数是未定义行为。

强大

MiniScript强大，是因为你不但可以创建对象，还可以使用原型继承的方式来拓展对象。基于原型的面向对象程序设计是面向对象的一个流派，由self语言开创，现在比较流行的语言里面，JavaScript和Lua采用这种设计方式。

这一段代码用原型继承模拟了传统的class - object的继承，创建了两个`Student`对象，这两个`Student`各自有各自的`name`成员，但是共享`teacher`成员。

```
Student = {
    "teacher": "fengyan",
    "new": function(name: str){
        student = {}
        student.name = name
        student.prototype = Student
        return student
    },
    "setName": function (self: table, name: str) {
        self.name = name return 0
    },
    "setTeacher": function (self: table, teacher: str) {
        self.teacher = teacher return 0
    }
}
boy1 = Student.new("Tom")
boy2 = Student.new("David")
boy1.setName(boy1, "Tomas")
print boy1.name + " " + boy1.teacher + " " + boy2.name + " " + boy2.teacher
boy2.setTeacher(boy2, "chenchun")
print boy1.name + " " + boy1.teacher + " " + boy2.name + " " + boy2.teacher
return 0
```

整体结构

MiniScript并不跑在真实机器上。相反，它运行在一个（经过了极大的简化的）Python虚拟机上。这个Python虚拟机是我从500 lines or less这本书里面展示的一个Python虚拟机修改过来的，原版大概有500多行代码，经过我的删减变成了300多行代码。MiniScript经过编译器的处理之后形成了Python字节码，然后虚拟机读入Python字节码运行。选择使用Python字节码是有原因的。从本质上来说，Python和Java, C#这些语言一样，都是编译执行的（尽管人们常常认为Python是解释执行的）。Python的编译器会将Python程序编译成字节码，然后进入虚拟机运行。而Python虚拟机和传统的jvm以及.NET不同的地方在于它更加高层，离真实机器更远，提供了更加丰富的字节码指令，不仅包括算术运算，赋值，还包括建立循环，调用函数等。将语言编译到Python字节码有利于简化程序编译的步骤，利用更少的代码实现更丰富的功能。

MiniScript语言的实现总共只有1300行左右的代码。它的核心代码包含300多行的语法制导翻译，300多行的代码生成和300多行的虚拟机。整个MiniScript语言的程序架构是这样的：

编译器将程序读入之后，程序会经过词法分析转换成token流，token流经过语法制导翻译形成语法树，语法树经过代码生成形成虚拟机代码。

常见问题

你们是用什么语言做的？

我们用的是Python

你们**lex**和**yacc**是怎么解决的？

我们用了一个Python里面的lex和yacc实现，叫做ply。

我们的要求是写一个类**C**，**Tiger**或者**Pascal**的语言，而你们实现的语言是**JavaScript**

是的，我们实现的语言很像JavaScript。但是JavaScript也是类C的。

再有，之所以我们不做类似C语言和Tiger语言的语法，是因为我们不想抄课本代码。而且这两种语言的语法都有设计上的缺陷。比如，C语言规定“一个变量的声明要仿照它的使用”。就像下面这样

```
int *p[3];
```

这一段代码的含义是：*p[3] = int ，所以p[3] = a pointer to int， 所以p = an array of pointer to int。这种“解方程”式的声明变量语法强行使一个变量的类型信息被这个变量本身拦腰截断，给parsing过程带来了很大的困难。实际上，如果一个语言的语法包含对变量的类型声明的话，目前公认的最好方式是把类型放在后面，大多数现代语言都使用了这种做法，包括TypeScript, Rust 和 Go。

我们在设计语言的时候，时刻关注语法的简洁，优雅，易于parse，最终选择了类似JavaScript的语法。它的语法成分不多，简单也不失灵活，更重要的是方便parse。

最后，就像评价一个人的时候不应该过分关注他的面貌，而应该关注他的学识素养和道德水准一样，我们认为编译原理的大作业也不应该重点关注“实现了一个什么样的语言”，而在于“做的这个语言里面词法分析，语法分析，语义分析和代码生成做得怎么样”。

你们做的是一个解释器还是编译器？

我们整个程序的执行流程是：用户输入的程序被翻译成虚拟机上的代码，然后虚拟机运行这一段代码。换句话说，我们做的这个语言和Java,C#没有什么区别，它的基本运行方式都是先从程序生成字节码，再在虚拟机上运行字节码。不同之处在于我们的字节码设计参考的是Python字节码，它比JVM字节码和.NET上面的字节码更加高层，提供了更加高的抽象。

我们对“编译”和“解释”的理解是，只要一个语言的运行过程中有把程序代码翻译成某种字节码或者机器指令的过程，这个语言就被编译了。从这个角度来看，包括JavaScript和Python在内的大部分语言都是编译执行的，只有少数如Shell之类的语言是纯解释执行的。

词法规则

关键字

根据我们的语言所需，定义了如下关键字，按照ply的lex语法规定，将关键字保存在reserved_word中。

```
reserved_word = {
  'for': 'FOR',
  'while': 'WHILE',
  'if': 'IF',
  'else': 'ELSE',
  'print': 'PRINT',
  'and': 'AND',
  'or': 'OR',
  'not': 'NOT',
  'break': 'BREAK',
  'function': 'FUNCTION',
  'return': 'RETURN'
}
```

符号标记

根据ply的语法规则，将语法中所用到的符号定义对应的标记符号TOKENS。标记TOKENS定义在最前面，以列表的形式存储。每种TOKEN用一个正则表达式规则来表示，每个规则需要以"t_"开头声明，表示该声明是对标记的规则定义。对于简单的标记，可以直接定义：

```
  t_PLUS = r'\+'
  t_MINUS = r'-'
  t_TIMES = r'\*'
  t_DIVIDE = r'/'
  t_LEFT_BRACE = r'\{'
  t_RIGHT_BRACE = r'\}'
  t_LEFT_BRACKET = r'\['
  t_RIGHT_BRACKET = r'\]'
  t_LEFT_PAREN = r'\('
  t_RIGHT_PAREN = r'\)'
  t_ASSIGN = r'='
  t_SEMICOLON = r';'
  t_EQUAL = r'=='
  t_NOT_EQUAL = r'!='
  t_GREATER = r''
  t_LESS = r''
  t_GREATER_EQUAL = r'='
  t_LESS_EQUAL = r'='
  t_COMMA = r','
  t_COLON = r'\:'
  t_DOT = r'\.'
```

对于需要执行动作的符号标记，如整数、实数、字符串和ID等TOKENS，将规则写成一个方法。方法总是需要接受一个LexToken实例的参数，该实例有一个t.type的属性（字符串表示）来表示标记的类型名称，t.value是标记值（匹配的实际的字符串）。方法可以在方法体里面修改这些属性。但是，如果这样做，应该返回结果token，否则，标记将被丢弃。
定义如下：

```
def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_REAL(t):
    r'[0-9]*\.[0-9]+'
    t.value = float(t.value)
    return t

def t_STRING(t):
    r'"[\^"]*"'
    t.value = t.value[1:len(t.value) - 1]
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved_word.get(t.value, 'ID') # Check for reserved words
    return t
```

匹配规则顺序

在lex内部，lex.py用re模块处理匹配模式，匹配顺序如下：

1. 所有由方法定义的标记规则，按照他们的出现顺序依次加入

2. 由字符串变量定义的标记规则按照其正则表达式长度倒序后，依次加入

语法规则

MiniScript语法的基本组成元素是statement（语句）和expression（表达式），这两者的区别在于语句运行后没有返回值，而表达式运行后有返回值。比如

```
a = 1
```

是一条赋值语句，没有返回值。注意，我们的语言和C语言不同。在C语言里面赋值运算是有返回值的表达式，这是一个重要的设计缺陷，因为这使得人们可以写出类似于

```
if (a = 0)
{
     //body
}
```

的代码，造成了大量微妙的bug。

表达式

表达式分为以下几类：常量表达式，标识符表达式，一元运算表达式，二元运算表达式，比较表达式，括号表达式，函数调用表达式，数组表达式和table表达式。它们的共同特点是：都返回一个值。除了函数调用表达式之外，其他的表达式都是没有副作用的。一种设计思路是不允许函数调用修改任何外部变量，包括通过引用传递给它的参数，这使得函数调用表达式也没有副作用，也就是所谓的“纯函数式”语言（比如Haskell），但是这种设计将大大减弱语言的能力。

常量表达式

常量表达式就表示一个常量，所谓的常量包括整数，实数，字符串和函数。比如数字`3`是一个常量表达式，函数定义式

```
function(a: int, b: int){
    return a + b
}
```

也是一个常量表达式。注意，我们的语言里面“函数”和通常的值没有什么区别。相应的语法如下

```
expression : INT
expression : REAL
expression : STRING
expression : FUNCTION LEFT_PAREN parameter_list RIGHT_PAREN LEFT_BRACE statement_list
RIGHT_BRACE
```

标识符表达式

标识符表达式就表示一个用户定义的标识符，比如说在

```
age + 8
```

里面，用户自定义标识符`age`就是一个标识符表达式。语法如下

```
expression : ID
```

一元表达式

一元运算表达式表达一个一元运算，比如说

```
not a
-(1 + 2)
```

相应的语法如下：

```
expression : MINUS expression %prec UNARY_MINUS
             | NOT expression
```

二元表达式

二元表达式表达一个二元运算，不仅包括加减乘除，还包括下标运算和点号取成员运算

```
a + b
array[3]
student.age
```

相应的语法如下

```
expression : expression PLUS expression
              | expression MINUS expression
              | expression TIMES expression
              | expression DIVIDE expression
              | expression AND expression
              | expression OR expression
expression : expression LEFT_BRACKET expression RIGHT_BRACKET  # binary subscrpt
expression
expression : expression DOT ID # binary dot expression
```

比较表达式

比较表达式表达一种比较关系，也属于二元表达式，但是由于在代码生成方面有所不同和二元表达式分开

比较表达式的例子

```
a < b
c > (d + e)
```

相应的语法如下

```
expression :   expression EQUAL expression
                | expression NOT_EQUAL expression
                | expression GREATER expression
                | expression LESS expression
                | expression GREATER_EQUAL expression
                | expression LESS_EQUAL expression
```

括号表达式

括号表达式就是把一个表达式括号括起来，表达优先级

```
 expression : LEFT_PAREN expression RIGHT_PAREN
```

函数调用表达式

函数调用表达式就是表达一个函数调用，语法如下

```
expression : expression LEFT_PAREN expression_list RIGHT_PAREN
```

数组表达式

数组表达式表达一个数组的生成，比如说

```
[1, 2, "hello", [2, 3]]
```

数组表达式的语法如下：

```
expression : LEFT_BRACKET expression_list RIGHT_BRACKET
```

table表达式

table表达式表示一个`table`，这是我们这个语言里面的核心数据结构。比如说下面这些都是table表达式。

```
{"name": "zouyao", "age": 21}
```

```
{
     "msg": "hello",
     "printMsg": function(){
          //body
     }
}
```

table表达式的语法如下：

```
map_element_expression : expression COLON expression
map_element_list :  map_element_expression
                  | map_element_expression COMMA map_element_list
                  |
expression : LEFT_BRACE map_element_list RIGHT_BRACE
```

表达式列表

有的时候我们需要一个数据结构来表达一串用逗号分割的表达式，比如在向函数传递参数的时候。这时我们就需要表达式列表这个数据结构，它表达一串用逗号分割的表达式。比如

```
someFunction(1, "someValue", a + b)
```

在这里，小括号中间的

```
1, "someValue", a + b
```

构成了一个表达式列表，这个表达式列表的前两个元素是常量表达式，后一个元素是二元表达式。表达式列表的语法如下：

```
  expression_list :  expression COMMA expression_list
                   |  expression
                   |
```

优先级与结合性

我们可以发现，在上面的推导式中，我们都是直接使用expression作为推导式的元素，这种文法是具有二义性的，需要利用优先级和结合性规则来消除二义性。优先级结合性规则和C语言相同，这里不再赘述。

语句

在我们的语言里面，语句是一种执行一串操作（通常有副作用）但是没有返回值的语法元素。我们的语言中，语句包含这几类：条件语句，while语句，for语句，赋值语句，下标赋值语句，print 语句，表达式语句，break语句和return语句。用一个大括号包起来的多个语句形成了一个复合语句，它也是一种语句。注意两点，第一点，我们的语言中，语句结尾不需要加分号；第二点，我们的语言提供的循环控制语句只有break，没有continue更没有goto。实践经验表明，continue和goto会让语言的控制流显得混乱，不应该提倡使用。

```
  statement : compound_statement
            |  branch_statement
            |  while_statement
            |  for_statement
            |  assign_statement
            |  assign_subscr_statement
            |  print_statement
            |  expression_statement
            |  break_statement
            |  return_statement
```

条件语句

条件语句就是if 和else，语法如下

```
  branch_statement : IF LEFT_PAREN expression RIGHT_PAREN statement %prec INCOMPLETE_IF
                   | IF LEFT_PAREN expression RIGHT_PAREN statement ELSE statement
```

INCOMPLETE_IF的作用是利用优先级规则来消除shift-reduce冲突。

while语句

while语句就是while语句，语法如下：

```
 while_statement : WHILE LEFT_PAREN expression RIGHT_PAREN statement
```

for语句

for语句就是for语句

```
 for_statement : FOR LEFT_PAREN statement SEMICOLON expression SEMICOLON statement
RIGHT_PAREN statement
```

注意，在for语句的括号中，第一个语法元素是语句(statement)，第二个语法元素是表达式(expression)，第三个语法元素是语句(statement)，这和for语句的一般用法一致，因为for语句的通常用法是

```
for (i = 0; i < n; i = i + 1){
    //body
}
```

先做一个有副作用的赋值，再做一个需要返回值的判断，最后做一个有副作用的赋值

赋值语句

赋值语句的左边是一个标识符，右边是一个表达式

```
  assign_statement : ID ASSIGN expression
```

由于赋值是语句而不是表达式，不返回值，所以你不可以写出C语言里面的连续赋值，比如说

```
a = b = 1
```

可以写连续赋值是C语言把赋值当做表达式的唯一优点，然而这种写法并不会让你的程序节省几行代码，所以优势也不大。

下标赋值语句

下标赋值语句的左边是一个下标运算操作，右边是一个表达式，比如说

```
student["age"] = 21
```

当下标运算操作的中括号里面是一个字符串常量的时候，可以用点运算代替下标运算。比如说

```
student.age = 21
```

上面两个语句的效果是一模一样的。

下标赋值语句的语法如下

```
assign_subscr_statement : expression LEFT_BRACKET expression RIGHT_BRACKET ASSIGN
expression
assign_subscr_statement : expression DOT ID ASSIGN expression
```

break语句

break就是break，没什么好说的

```
break_statement : BREAK
```

return语句

return就是return

```
return_statement : RETURN expression
```

注意，由于我们这个语言的限制，每一个函数都必须在结尾写一个return语句。注意，首先你必须写return语句；然后，你必须写在函数结尾。否则即使编译可以通过，运行时也会发生不可预知的错误。

表达式语句

表达式语句就是一个表达式单独成为一个语句

```
expression_statement : expression
```

表达式语句是必须的，因为在有的时候，我们调用一个函数不是为了它的返回值而是为了它的副作用，这时我们一个语句里面就只有一个函数调用表达式。但是表达式语句的出现给我们这个语法带来了歧义。比如说，考虑如下这一段程序

```
b - 3
```

到底是parse成`b`一个表达式语句，`-3`一个表达式语句，还是`b - 3`一整个parse成一个表达式语句呢？由于yacc在出现shift-reduce冲突的时候默认的行为是shift，所以运行出来的结果是后者。这也就意味着你可能会写出这样的代码

```
b
-f(3)
```

期望编译器parse出两个表达式语句。然而编译器会把`b - f(3)`整个当做一个表达式语句parse出来。

print语句

为了能让程序展现出“可以运行”的样子，我们增加了一个print语句

```
print_statement : PRINT expression
```

复合语句

多个语句被一个大括号括起来会形成复合语句

```
compound_statement : LEFT_BRACE statement_list RIGHT_BRACE
```

复合语句和table表达式同时使用会使得我们的语法产生一个歧义：当用户写出一个空的

```
{}
```

的时候，应该把它判断成是一个空的复合语句还是一个空的table表达式呢？这个歧义其实无关紧要，因为不管parse成什么，一对空的大括号对于整个程序而言一点微小的影响都没有。

语句列表

语句列表就是一串语句连起来。语句之间不需要分号来隔开

```
statement_list : statement statement_list
                    |
```

语法分析的入口

语法分析的入口是program推导式，它就是一个statement_list

```
program : statement_list
```

函数

为了支持函数，我们增加了一些其他的语法元素。首先需要明确的是，在我们的语言里面，函数就是一个常量。我们定义一个函数`add`的语法是

```
add = function(a: int, b: int){
    return a + b
}
```

你需要注意的是，function关键字后面并没有写函数名字。在我们的语言里面，函数都是匿名定义的，所以或许把关键字function换成lambda更加合理。定义一个函数的语义和在C++以及JavaScript里面定义一个lambda表达式没有区别，就是你创造了一个类型为'function'的常量，然后把这个常量赋值给一个叫add的变量。函数常量的语法是：

```
expression : FUNCTION LEFT_PAREN parameter_list RIGHT_PAREN LEFT_BRACE statement_list
RIGHT_BRACE
parameter_list :  parameter COMMA parameter_list
               | parameter
               |
parameter : ID COLON type
type : TYPE_INT
     | TYPE_REAL
     | TYPE_STRING
     | FUNCTION
     | TYPE_TABLE
```

其中，`TYPE_INT`, `TYPE_REAL`, `TYPE_STRING`, `FUNCTION_TYPE_TABLE`分别表示关键字`int`， `real`, `str`, `function` 和 `table`。

语法制导翻译

语法制导翻译就是在语法分析的过程中生成语法树。我们将语法分析，语法制导翻译和语义分析一起做，因为我们这个语言的语法十分简单，不需要做特别复杂的中间过程。

语法树

我们提到过，整个语言有两个语法元素：表达式和语句。所以我们的语法树有两种节点，语句节点和表达式节点。但是，有的时候我们需要处理一些list,比如说复合语句里面有`statement_list`，而函数传递参数的时候有`expression_list`。所以我们又定义了一个列表节点

```
class StatNode:
    def __init__(self):
        self.type = None  # type: str
        self.children = None  # type: OrderedDict

    def emit_code(self) -> str:
        return NotImplemented


class ExpNode:
    def __init__(self):
        self.type = None  # type: str
        self.children = None  # type: OrderedDict

    def emit_code(self) -> str:
        return NotImplemented


class ListNode:
    ...
```

语法树上所有的节点，必定是这三个类的子类的实例。我们可以看到每一个类都有一个type字段，这是为了表达某个节点属于哪个子类。同时，每一个类都有一个`children`字段。`StatNode`和`ExpNode`的`children`是一个有序字典，而`ListNode`的`children`是一个列表。每一个类都有一个`emit_code`方法，用于做代码生成。

一般情况下，一个语法生成式就对应一个树节点的生成。我们看几个例子。比如说，我们以整数常量表达式为例

```
class ConstExpNode(ExpNode):
    def __init__(self, index: int):
        super().__init__()
        self.type = 'const'
        self.children = {'index': index}

def p_expression_int(p):
    """
    expression : INT
    """
    index = # get the index of p[1]
    p[0] = ConstExpNode(index)
```

我们发现，`ConstExpNode`这个节点里面的`children`是一个index。这个index表示这个const量在某一个列表中的下标。为什么我们不直接把常量值本身作为children呢？对于整数，直接以常量值作为children是可以的。但是如果这个常量是一个字符串甚至一个函数的话，存一个index是更好的选择。

在语法分析器parse到一个整数的时候，它以某种我们会在下一节讲述的方法得到这个整数常量的index，然后构造一个`ConstExpNode`。

同样的事情发生在标识符表达式上。注意到`IDExpNode`里面存放的也是这个标识符的某种下标，而不是这个标识符本身

```
class IDExpNode(ExpNode):
    def __init__(self, index: int):
        super().__init__()
        self.type = 'id'
        self.children = {'index': index}

def p_expression_id(p):
    """
    expression : ID
    """
    index = # get the index of p[1]
    p[0] = IDExpNode(index)
```

而当语法分析器看到一个二元表达式的时候，它就会按照语法规则生成一个`BinaryExpNode`，就像下面这样

```
class BinaryExpNode(ExpNode):
    def __init__(self, left: ExpNode, right: ExpNode, operator: str):
        super().__init__()
        self.type = 'binary'
        self.children = OrderedDict([('operator', operator), ('left', left), ('right',
right)])

def p_binary_expression(p):
    """
    expression : expression PLUS expression
               | expression MINUS expression
               | expression TIMES expression
               | expression DIVIDE expression
               | expression AND expression
               | expression OR expression
    """
    p[0] = BinaryExpNode(p[1], p[3], p[2])

def p_binary_subscr_expression(p):
    """
    expression : expression LEFT_BRACKET expression RIGHT_BRACKET
    """
    p[0] = BinaryExpNode(p[1], p[3], '[]')
```

看上去更加复杂的结构，比如说一个for循环语句，也是这么做的。

```
class ForStatNode(StatNode):
    def __init__(self, init_stat: AssignStatNode, condition_exp: BinaryExpNode,
                 loop_stat: AssignStatNode, body_stat: StatNode):
        super().__init__()
        self.type = 'for'
        self.children = OrderedDict([('init', init_stat), ('condition',
condition_exp),
                                     ( 'loop', loop_stat), ('body', body_stat)])

def p_for_statement(p):
    """
    for_statement : FOR LEFT_PAREN statement SEMICOLON expression SEMICOLON statement
RIGHT_PAREN statement
    """
    p[0] = ForStatNode(p[3], p[5], p[7], p[9])
```

整个程序是一个`ProgramNode`，为了方便，虽然语义上`ProgramNode`不应该是`StatNode`的子类，我们还是让它继承自`StatNode`。它的`children`是一个`StatListNode`。

```
class ProgramNode(StatNode):
    def __init__(self, stat_list: StatListNode):
        super().__init__()
        self.type = 'program'
        self.children = {'stat_list': stat_list}

def p_program(p):
    """
    program : statement_list
    """
    p[0] = ProgramNode(p[1])
```

列表节点的生成也很简单。我们这里有三种列表节点，`StatListNode`，`ExpListNode`和`MapElementListNode`分别对应`statement_list`，`expression_list`和`map_element_list`。`map_element_list`是用于table这种数据结构的。我们以`StatListNode`为例讲解。

```
class ListNode:
    def __init__(self, init_item=None):
        self.type = None  # type: str
        if not init_item:
            self.children = []
        else:
            self.children = [init_item]

    def emit_code(self):
        ...

    def prepend_item(self, item):
        self.children.insert(0, item)

class StatListNode(ListNode):
    def __init__(self, init_item=None):
        super().__init__(init_item)
        self.type = 'statement-list'

def p_statement_list(p):
    """
    statement_list : statement statement_list
                   |
    """
    if len(p) == 1:
        p[0] = StatListNode()
    else:
        p[0] = p[2]
        p[0].prepend_item(p[1])
```

我们发现，语法分析器parse `statement_list` 的方法就是：看到一个新的 `statement` 就利用 `prepend_item` 把它加入到原有的 `StatListNode` 的 `children` 里面。语法分析器分析 `expression_list` 和 `map_element_list` 的过程也基本相似。

由于其他表达式和语句的语法制导翻译过程基本类似，这里我就不再赘述了。下面我来介绍我们编译器中关键的一部分：作用域规则。这涉及到上下文有关的语法制导翻译，这也是我们整个编译器里面唯一的语义分析。事实上，我们这个语言不在编译时检查类型，所以编译时做的唯一语义分析就是确定作用域。

作用域规则及其实现

我们这个语言以函数作为作用域的基础。也就是说，一个函数是一个词法作用域。同时，我们的语言允许用户在函数内部定义函数（因为函数实际上就是一个常量，在函数内部定义一个常量一点也不稀奇古怪）。在函数内部定义的函数可以访问函数外部作用域的变量。我们举几个例子来说明这一点

```
f = function(n: int) {
    if (n == 1){
        ret = 1
    }else {
        ret = 0
    }
    return ret
}
```

上面是一个函数的定义。我们可以发现，`ret`这个变量是在if语句里面定义的，但是你可以在if语句外面使用它。因为我们的MiniScript是以函数作为作用域单位的，在一个if语句块里面定义变量和在整个函数里面定义变量没有什么区别。

```
f = function(base: int, n: int) {
    g = function(n: int) {
        if(n > base) {
            print "larger"
        }else {
            peint "smaller"
        }
        return 0
    }
    g(n)
    return 0
}
f(2, 3)
return 0
```

这是一个完整的MiniScript程序，这个程序里面我们定义了一个函数，把这个函数赋值给变量`f`，然后在这个函数里面定义了另一个函数`g`。注意，函数`g`使用了函数f的本地变量`base`。我们的语言里面，内部函数可以使用外部函数的本地变量。

当你在某一个函数`f`里面使用一个标识符`n`的时候，如果仅仅是读它的值，没有写诸如`n = 2`之类的赋值语句，我们的语言会先在`f`里面找这个`n`。如果没有找到`n`而`f`定义在另一个函数`g`里面，它会在`g`里面找这个标识符`n`。如果还没有找到则继续向外寻找，一直到找到全局区域为止。如果全局区域里面也没有这个`n`，则会报一个错误。相反，如果你对`n`做了赋值，我们的语言会先去寻找`n`的定义，从`f`函数内部找到它外部的函数`g`，一直找到全局区域。如果找到了，就会对找到的那个`n`进行赋值。如果没找到，就会在当前函数`f`的作用域内创建一个标识符`n`，进行赋值。

这样一种作用域规则是如何实现的呢？这就涉及到我们的编译器传递给虚拟机的字节码。编译器传给虚拟机的字节码是这样定义的：

```
class CodeObj:
    def __init__(self, code, const_list, name_list):
        self.code = code
        self.const_list = const_list
        self.name_list = name_list
```

`code`表示编译出来的代码，`const_list`表示常量列表，`name_list`表示名字列表（也就是标识符列表）。编译器传给虚拟机的就是这样一个`CodeObj`对象。同时，我们说函数是一个值，表示一个函数的值是这么定义的：

```
class Function:
    def __init__(self, parameter_list: ParameterList, code_obj: CodeObj,
lexical_depth: int):
        self.parameter_list = parameter_list
        self.code_obj = code_obj
        self.lexical_depth = lexical_depth
```

函数的三个字段分别是参数列表，目标代码和函数的词法深度。`code_obj`字段的类型就是上面的`CodeObj`类。也就是说，每一个函数里面也有一个`CodeObj`对象。这就意味着整个程序里面有一个`name_list`，每一个函数里面也有一个`name_list`。`name_list`的类型就是一个`str`数组。比如说，在上面那一段程序中，全局的`name_list`是`['f']`，这是一个被赋值了一个函数的变量；`f`的`name_list`是`['base', 'n', 'g']`，`g`的`name_list`是`['n']`.

在上一节我们说过，`IDExpNode`的`children`是一个index，这个index就包含了两部分信息：一是这个标识符处于哪个作用域里面，二是这个标识符位于那个作用域的name_list的什么位置。我们用一个tuple来表示这两个信息，将这个tuple命名为(depth, index)，其中depth是这个标识符所处的作用域的词法深度。词法深度就是这个标识符所在的函数的嵌套层次。比如说如果你在程序里面定义了函数f，f里面定义了函数g，那么f的嵌套层次是1，g的嵌套层次是2.于是，所有在全局区定义的标识符的词法深度都是0，在f里面定义的标识符的词法深度是1，在g里面的是2.举一个例子，在上面的例子里面，g引用了n和base这两个标识符，其中n是这个函数g的本地变量，因为g在嵌套层次的第二层，所以n的作用域词法深度是2.因为base是g外面的函数f的本地变量，而f在嵌套深度的第一层，所以base的作用域词法深度是1.

这个tuple的第二个字段是index，也就是该标识符在它所在的那个作用域的`name_list`里的下标。比如说，假设g的`name_list`是`['n']`，那么g在引用n这个标识符的时候，形成的tuple就是(2, 0),表示n是词法深度为2的那个作用域（也就是g的作用域）的`name_list`的第0个元素。而g里面引用base的时候，所形成的tuple是(1, 0)，表示base这个标识符是词法深度为1的那个作用域（也就是f的作用与）的nale_list的第0个元素。

`IDExpNode`的children就是由这样一个tuple变化过来的。因为字节码不允许带有一个tuple作为参数，所以我我们把这个tuple变成一个整数存到`IDExpNode`里面，具体的方法是`IDExpNode.children['index'] = tuple.depth << 8 + tuple.index`。

为了能在parse的时候获得某个标识符的词法深度和他在对应的作用域的name_list里面的下标，我们建立了一个叫做`ScopeManeger`的类。这个类管理这这个程序里面所有的`name_lis`t。当parser看到一个函数的时候，它就为那个函数建立一个新的`name_list`。同时，这个类还为每一个`name_list`维护着一个`prev_name_list`，表示它的上一层作用域的`name_list`。比如说，在我们的例子里面，函数g的`prev_name_list`就是函数f的`name_list`，而函数f的`prev_name_list`就是全局的`name_list`。这样，我们在某个函数里面看到一个变量的时候，可以利用这些信息确定这个变量来自哪一个scope。注意我们的语言不支持JavaScript风格的闭包，所以对于每一个标识符，只需要记录它位于哪个词法深度和它位于他所在的作用域的`name_list`的哪个位置就可以了。

虚拟机有两种方法在运行时根据这些信息寻找到所需要的变量，第一种是为每一个函数frame维护一个static link，第二种是维护一个全局的display数组。我们采用的是第二种方法。关于display数组，我记得我们有一道习题是有关它的，所以这里不再赘述。

代码生成

我们的编译器生成的是（略微修改了的）Python字节码。如果你从未接触过Python字节码，那么我推荐你先看一看 500 lines or less 里面的a Python interpreter written in Python，里面有一些关于Python字节码的入门。Python官方文档是关于Python字节码的权威资料。以下内容假设你了解一定的Python字节码。

代码定义

我们使用的字节码是一个略微修改了的Python字节码的一个子集。注意，Python虚拟机是一个基于堆栈的虚拟机，在以下内容里面TOS指的是堆栈顶端的元素，TOS1指的是堆栈顶端第二个元素，以此类推。

堆栈操作

- `POP_TOP`

  移除栈顶元素

一元运算

- `UNARY_NEGATIVE`

  实现 `TOS = -TOS`.

- `UNARY_NOT`

  实现`TOS = not TOS`.

二元运算

- `BINARY_MULTIPLY`

  实现 `TOS = TOS1 * TOS`.

- `BINARY_TRUE_DIVIDE`

  实现 `TOS = TOS1 / TOS`.

- `BINARY_ADD`

  实现`TOS = TOS1 + TOS`.

- `BINARY_SUBTRACT`

  实现 `TOS = TOS1 - TOS`.

- `BINARY_SUBSCR`

  实现 `TOS = TOS1[TOS]`.

- `BINARY_AND`

  实现 `TOS = TOS1 & TOS`.

- `BINARY_OR`

  实现 `TOS = TOS1 | TOS`.

- `COMPARE_OP`(opname)
  对TOS和TOS1进行布尔比较，*opname*是所使用的比较运算符的下标。比较结果放回TOS

循环

- `SETUP_LOOP`(target)
  建立一个循环，这个循环的终点在*target*处。之所以保留这个target是为了break的时候知道应该break到哪里去。

- `BREAK_LOOP`

  break语句终止循环

- `POP_BLOCK`

  从block栈中移除一个block

存和取

- `STORE_NAME` (*namei*)

  实现 `name = TOS`. *namei* 指的是这个标识符的下标，包含上面提到的作用域信息和index信息

- `LOAD_CONST` (*consti*)

  把 `const_list[consti]` 放到栈顶

- `LOAD_NAME` (*namei*)

  把 `name_list[namei]` 对应的那个值推到栈顶

- `STORE_SUBSCR`
  实现 `TOS1[TOS] = TOS2`.

构造table

- `BUILD_LIST` (*count*)

  从堆栈里面弹出*count*个元素，组成一个table`{..., count-2: TOS1, count-1: TOS}`放回栈顶

- `BUILD_MAP` (*count*)

  在堆栈上面创建一个新的table放回栈顶。创建之前从栈上推出*2*count*个元素，这样形成的table是`{..., TOS3: TOS2, TOS1:TOS}`.

跳转

- `POP_JUMP_IF_TRUE` (*target*)

  如果说TOS是true，就跳转到 *target*. TOS 被弹出。

- `POP_JUMP_IF_FALSE` (*target*)

  如果TOS是false，就跳转到*target*. TOS 被弹出.

- `JUMP_ABSOLUTE` (*target*)
  跳转到*target*

函数

- `CALL_FUNCTION`
  调用函数，函数放在TOS处。先弹出TOS，得到函数的定义信息，知道这个函数拥有argc个参数，然后从栈里面弹出argc个元素，运行这个函数。返回值放到TOS
- `RETURN_VALUE`
  函数返回，把TOS返回给调用者

交互

- `PRINT_EXPR`

  把TOS print出来。在print 语句中我们用到了这个字节码

代码生成操作

为堆栈式虚拟机生成代码非常简单，基本上只需要对语法树进行遍历就可以了。我们说过，所有的节点都继承自StatNode, ExpNode或者ListNode，而它们都定义了emit_code这个接口。所以，代码生成的过程，其实就是整个语法树emit_code的过程。

比如说，ConstExpNode和IDExpNode的代码生成是这样的

```
class ConstExpNode(ExpNode):
    def __init__(self, index: int):
        super().__init__()
        self.type = 'const'
        self.children = {'index': index}

    def emit_code(self):
        return 'LOAD_CONST {}'.format(self.children['index'])


class IDExpNode(ExpNode):
    def __init__(self, index: int):
        super().__init__()
        self.type = 'id'
        self.children = {'index': index}

    def emit_code(self):
        return 'LOAD_NAME {}'.format(self.children['index'])
```

它们只需要生成一条代码：把需要load的东西load到堆栈顶端。

稍微复杂一点的，比如说，二元表达式的代码生成是这样的

```
class BinaryExpNode(ExpNode):
    def __init__(self, left: ExpNode, right: ExpNode, operator: str):
        super().__init__()
        self.type = 'binary'
        self.children = OrderedDict([('operator', operator), ('left', left), ('right',
right)])

    def emit_code(self):
        [operator, left, right] = self.children.values()
        left_code = left.emit_code()
        right_code = right.emit_code()
        if operator == '+':
            operator_code = 'BINARY_ADD'
        elif operator == '-':
            operator_code = 'BINARY_SUBTRACT'
        ...
        else:
            raise ValueError('unrecognized binary operator {}'.format(operator))
        return left_code + '\n' + right_code + '\n' + operator_code
```

我只需要先生成左子表达式的代码，再生成右子表达式的代码，最后生成一个`BINARY_ADD`，或者`BINARY_SUBTRACT`就可以了。左右两个子表达式会生成它们自己的代码，它们的代码运行完成之后返回值一定在堆栈的顶端两个元素，这个时候再生成一个二元运算的代码就可以了。

再来看一些为语句生成代码的例子。比如说，为赋值语句生成代码

```
class AssignStatNode(StatNode):
    def __init__(self, id_index: int, exp: ExpNode):
        super().__init__()
        self.type = 'assign'
        self.children = OrderedDict([('id_index', id_index), ('exp', exp)])

    def emit_code(self):
        [id_index, exp] = self.children.values()
        exp_code = exp.emit_code()
        assign_code = 'STORE_NAME {}'.format(id_index)
        return exp_code + '\n' + assign_code
```

赋值语句只需要递归得先生成他所对应的那个表达式的求值代码，然后生成一个store_name就可以了。

又比如说，while语句的代码生成也是可以依葫芦画瓢。

```
class WhileStatNode(StatNode):
    def __init__(self, condition_exp: BinaryExpNode, body_stat: StatNode):
        super().__init__()
        self.type = 'while'
        self.children = OrderedDict([('condition', condition_exp), ('body',
body_stat)])

    def emit_code(self):
        [condition, body] = self.children.values()
        condition_code = condition.emit_code()
        body_code = body.emit_code()
        while_code = """SETUP_LOOP L{0}
L{1}:
{2}
POP_JUMP_IF_FALSE L{3}
{4}
JUMP_ABSOLUTE L{1}
L{3}:
POP_BLOCK
L{0}:""".format(next_label(), next_label(), condition_code, next_label(), body_code)
        return while_code
```

我们只需要建立一个让程序在循环里面跳转的框架，然后递归地生成条件判断和循环体的代码就可以了。`next_label()`是一个函数，返回下一个可以用的label值。比如说假设现在用到了label 4，那么生成的代码就会是

```
SETUP_LOOP L5
L6:
{condition_code}
POP_JUMP_IF_FALSE L7
{body_code}
JUMP_ABSOLUTE L6
L7:
POP_BLOCK
L5:
```

请注意千万不要在C++里面这样写代码，因为C++在函数调用时，参数的求值顺序是不确定的。Python确定了参数求值顺序，所以我们在这里可以少写很多代码。

代码生成基本上就是这样的过程，递归遍历语法树，然后递归生成代码即可。

运行环境

运行环境是一个极大的简化了的Python虚拟机。这个Python虚拟机的想法出自 500 lines or less 里面的a Python interpreter written in Python，虽然作者号称写了一个Python解释器，但是实际上他仅仅是写了一个Python虚拟机。也就是说，它只能运行Python字节码，缺少一般的解释器或者编译器所必备的词法分析，语法分析，代码生成等过程。你可以在它的github地址获取它的源代码。这份代码是MIT协议开源的，所以我在作业中使用它并不会带来版权问题。原版代码大约有500行，经过我的修改变成了大约350行。你可以先打开上面给出的这本书的链接阅读相关章节，了解它的实现思路。虚拟机部分不在编译原理的范畴之内，所以我只是简单讲一下我们做的修改。以下内容假设你已经看过书上相关章节并且阅读过配套代码。

- 为了支持原型继承，在对某个table取下标的时候，不但会检查这个table自己的成员，还会检查其prototype的成员。比如说，当你使用`student.age`的时候，如果在`student`这个table里面找不到age这个关键字，那么虚拟机会寻找student有没有prototype这个关键字，如果有就进入`student.prototype`寻找age这个关键字。
- 为了支持作用域规则，使用了display数组来保存在每一个词法深度上访问过的最后一个函数frame。
- 去掉了原代码里面有关异常处理的代码。我们没有做异常。
- 统一了数组和table。当用户写出[1, 2, 3]这样一个数组出来的时候，虚拟机会把它变成{0: 1, 1:2, 2:3}这样一个table。这样简化了我们的类型系统
- 加入了运行期类型检查