怎样写一个解释器

lichao06

May 20, 2015

Interpreter

什么是解释器

Interpreter?

Interpreter

什么是解释器

Interpreter?

解释器是一个不需要将代码编译成机器码，就可以直接解释、执行程序（脚本）语言的程序。
常见的有 ruby, python, php, bc

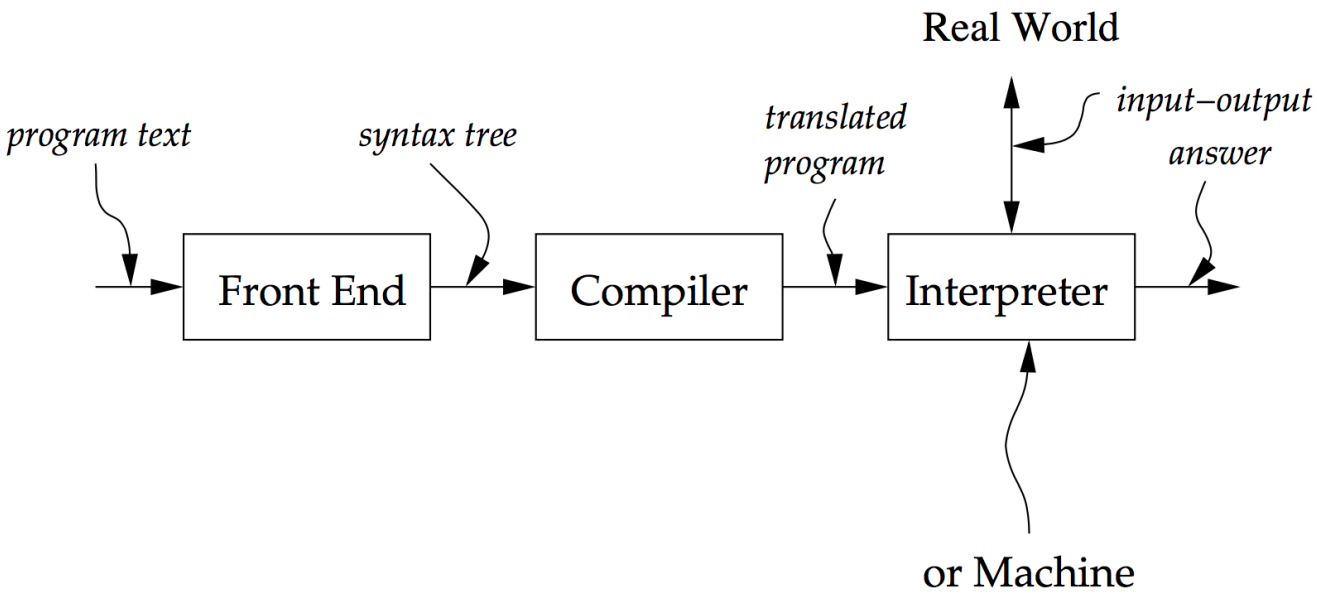
Interpreter?

解释器是一个不需要将代码编译成机器码，就可以直接解释、执行程序（脚本）语言的程序。
常见的有 ruby, python, php, bc

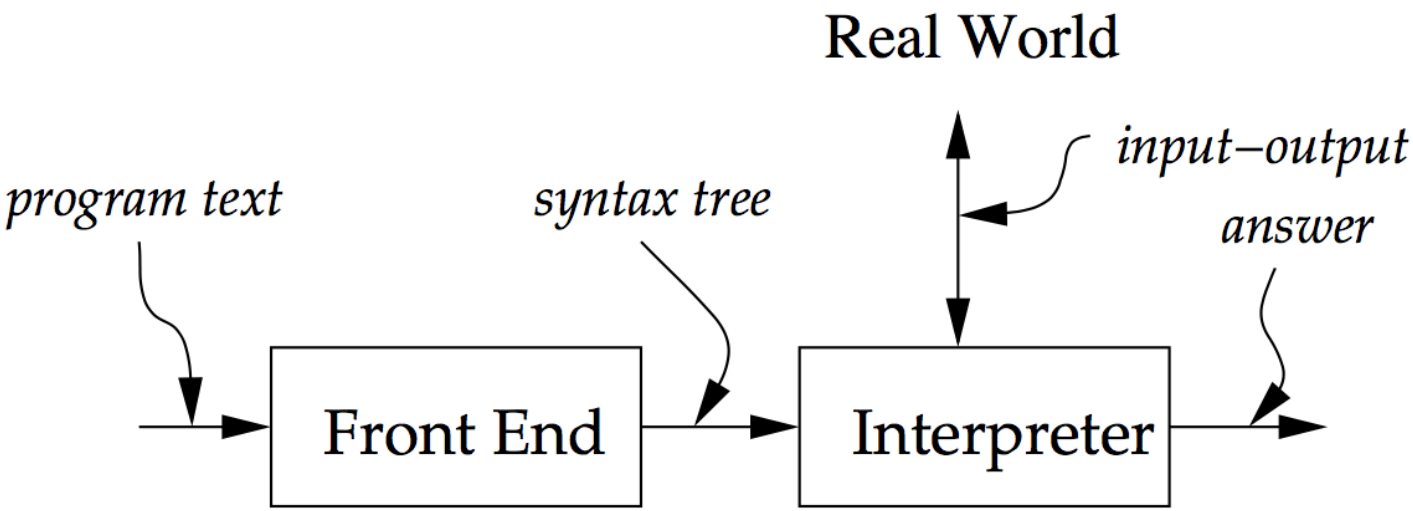
一般启动之后会有一个 提示符，等待用户输入。例如：

```
~ [liszt@liszts-MacBook-Pro]
# racket
Welcome to Racket v6.0.
>
```

常见的计算机语言需要使用编译器(e.g. gcc, g++)



解释器的步骤，稍微简单一些：



Front End

- 作用是把程序代码转换为 AST
- 通常分为 scanning & parsing 两部分
- Scanning 将“字符串”转换成一个 token 序列，token 可能是 单词、数字、注释……
- Parsing 将 token 序列组织为一个 AST
- Front End 读入的语言叫做“source language”
- Concrete Syntax

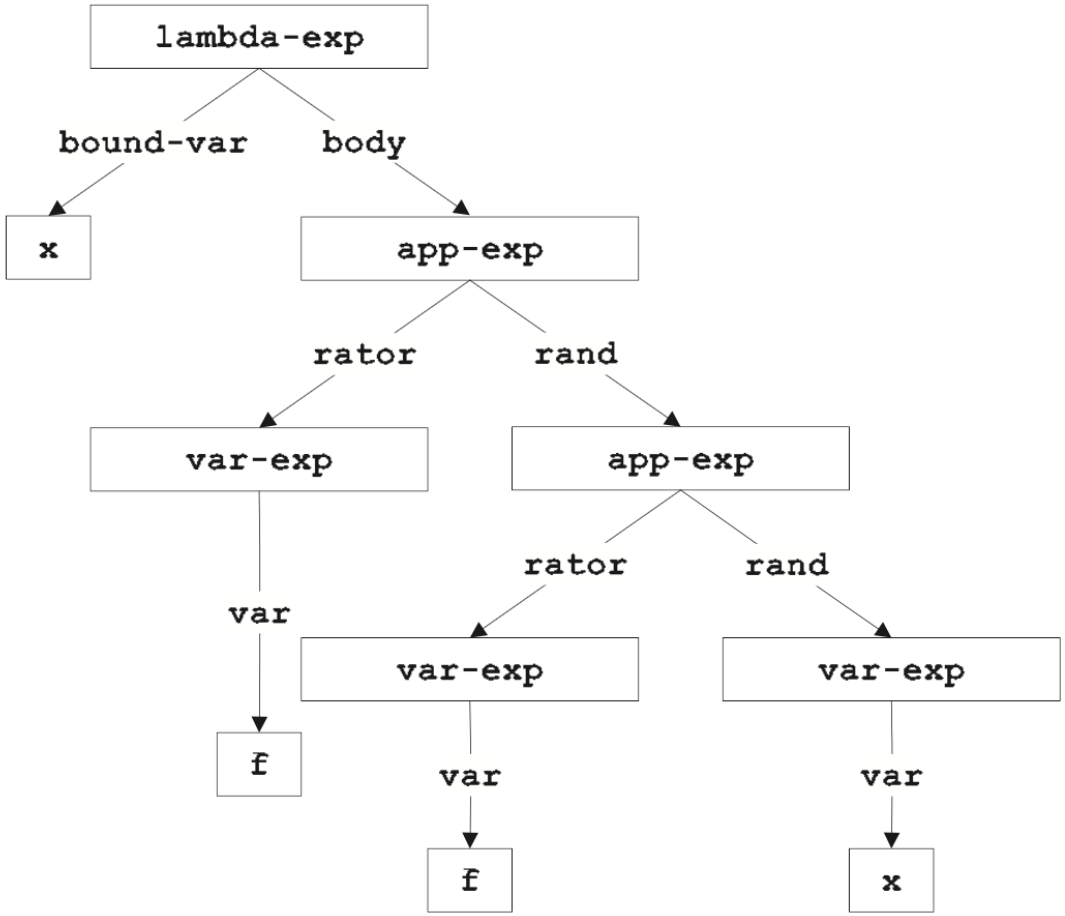
Scan & Parse

```
# racket
Welcome to Racket v6.0.
> (require "diff.scm")
> (just-scan "-(5, 8)")
'((literal-string34 "-" 1)
  (literal-string34 "(" 1)
  (number 5 1)
  (literal-string34 "," 1)
  (number 8 1)
  (literal-string34 ")" 1))
> (scan&parse "-(5, 8)")
(a-program (diff-exp (const-exp 5) (const-exp 8)))
```

Concrete Syntax

```
(lambda (x)
        (f (f x)))
```

Abstract Syntax



Interpreter 的工作流程是这样的：

- 输入是一个 AST
- Interpreter 根据 AST 的数据结构，执行后续操作
- 实现 Interpreter 的语言，叫做 “implementation language”

功能简介

写一个解释器

下面我们写一个简单的解释器，它只有一个功能： 读入用户输入的字符串，并且输出一个数字

类似于这样:

```
# racket
Welcome to Racket v6.0.
> (require "const.scm")
> (run "89")
(num-val 89)
> (run "64")
(num-val 64)
```

功能简介

写一个解释器

Syntax:

Program ::= Expression

a-program (exp1)

Expression ::= Number

const-exp (num)

Front End

写一个解释器

Syntax:

```
(define the-lexical-spec
  '((whitespace (whitespace) skip)
    (number (digit (arbno digit)) number)
    (number ("-" digit (arbno digit)) number)
    ))

(define the-grammar
  '((program (expression) a-program)
    (expression (number) const-exp)
    ))
```

使用 “sllgen”, 可以完成 字符串到 AST 的转换:

```
# racket
Welcome to Racket v6.0.
> (require "const.scm")
> (scan&parse "89")
(a-program (const-exp 89))
> (scan&parse "64")
(a-program (const-exp 64))
```

Interpreter

写一个解释器

解释器部分需要处理 Program 和 Expression

Pragram

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
                 (value-of exp1 (init-env))))))
```

Expression

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num)))))
```

执行结果

写一个解释器

下面执行几段代码：

```
# racket
Welcome to Racket v6.0.
> (require "const.scm")
> (run "89")
(num-val 89)
> (run "64")
(num-val 64)
```

增加功能

写一个解释器

下面我们为解释器增加 diff 、if 操作

定义数据类型：

```
(define-datatype expval expval?
  (num-val
   (value number?))
  (bool-val
   (boolean boolean?)))
```

增加功能

写一个解释器

由于计算需要在数字之间进行，if 需要布尔类型，所以需要它们到 AST 之间转换的方法

$$\begin{aligned}
&\textbf{num-val} && : Int \longrightarrow ExpVal \\
&\textbf{bool-val} && : Bool \longrightarrow ExpVal \\
&\textbf{expval->num} && : ExpVal \longrightarrow Int \\
&\textbf{expval->bool} && : ExpVal \longrightarrow Bool
\end{aligned}$$

diff

写一个解释器

Syntax:

Program ::= -(Expression , Expression)

diff-exp (exp1 exp2)

Front End

```
(define the-grammar
  '((program (expression) a-program)
    (expression (number) const-exp)
    (expression
     ("-" "(" expression "," expression ")")
     diff-exp)
    ))
```

Interpreter

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (diff-exp (exp1 exp2)
                (let ((val1 (value-of exp1 env))
                      (val2 (value-of exp2 env)))
                  (let ((num1 (expval->num val1))
                        (num2 (expval->num val2)))
                    (num-val
                     (- num1 num2)))))
)))
```

zero?, if

写一个解释器

Syntax:

Program ::= zero? (Expression)

zero?-exp (exp1)

Program ::= if Expression then Expression else Expression

if-exp (exp1 exp2 exp3)

Front End

```
(define the-grammar
  '((program (expression) a-program)
    (expression
     ("zero?" "(" expression ")")
     zero?-exp)

    (expression
     ("if" expression "then" expression "else"
         expression)
     if-exp)
    ))
```

zero?、if

写一个解释器

Interpreter

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (zero?-exp (exp1)
                 (let ((val1 (value-of exp1 env)))
                   (let ((num1 (expval->num val1)))
                     (if (zero? num1)
                         (bool-val #t)
                         (bool-val #f))))))
      (if-exp (exp1 exp2 exp3)
              (let ((val1 (value-of exp1 env)))
                (if (expval->bool val1)
                    (value-of exp2 env)
                    (value-of exp3 env))))
)))
```

执行结果

写一个解释器

下面执行几段代码：

```
# racket
Welcome to Racket v6.0.
> (require "diff.scm")
> (run "-(89, 64)")
(num-val 25)
> (run "if zero? (-(89, 64)) then 0 else 1")
(num-val 1)
```

Variable

写一个解释器

表达式中的变量，需要在一个“环境”去求值才有意义(变量的具体值存储在环境中)

一个环境有三个最基本的操作:

- empty-env 初始化一个 environment
- extend-env 绑定一对变量，并返回绑定后的 environment
- apply-env 从环境中读取变量的值

Let

写一个解释器

Syntax:

Program ::= let Identifier = Expression in Expression

let-exp (var exp1 body)

Front End

```
(define the-grammar
  '((expression
     ("let" identifier "=" expression "in" expression)
     let-exp)
    ))
```

Let

写一个解释器

Interpreter

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (let-exp (var exp1 body)
               (let ((val1 (value-of exp1 env)))
                 (value-of body
                           (extend-env var val1
                              env))))
)))
```

执行结果

写一个解释器

下面执行几段代码：

```
# racket
Welcome to Racket v6.0.
> (require "diff.scm")
> (run "let x = 89 in x")
(num-val 89)
> (run "let x = 0 in if zero?(x) then 0 else 1")
(num-val 0)
```

解释器语法

写一个解释器

汇总一下前面这个解释器支持的语法：

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

ToDo

写一个解释器

后续你可以继续为这个解释器添加以下功能，来得到一门基本可用的语言：

- 函数
- 递归
- 各种语法糖

QA

写一个解释器

ref:

- http://www.eopl3.com/
- https://mitpress.mit.edu/sicp/
- http://docs.racket-lang.org/eopl/index.html?q=