

写一个解释器

刘恩泽

2017-09-24

目录

什么是解释器

如何下手

半小时版本

实现一个解释器

最后

什么是解释器

解释器

- 同声传译
- 一段能够理解并执行 你的程序的 程序
 - 理解你的代码所表示的意图
 - 执行你的意图

代码的意图

- 赋值/定义

```
1 (setf a 1)
2 (defun plus (a b) (+ a b))
```

- 取值

a

- 执行

(plus 1 2)

语法 (S Expression)

- 原子 a, 1, "hello world"
- 表 (), nil, (a 1 2), (a . b), (a . nil), (a (b))

语义

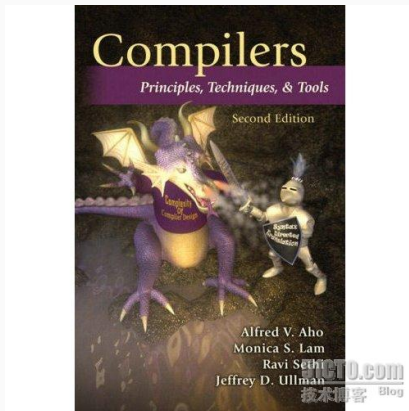
原子表达式 即 a, 1 等原子, 可直接求值或上下文中查找对应的值

复合表达式 函数

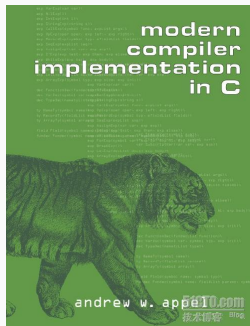
特殊形式 求值方式与函数不一致

如何下手

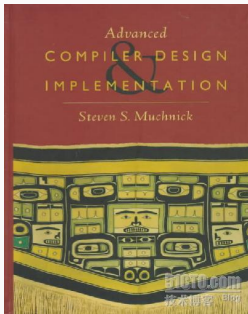
Dragon book, 中文名 编译原理



Tiger book, 中文名 现代编译原理-C 语言描述



Whale book, 中文名 高级编译器设计与实现



结束

- 好，分享结束，大家可以回去看书了.

结束

- 好，分享结束，大家可以回去看书了.
- 预计一年后，应该可以成功写出来了。

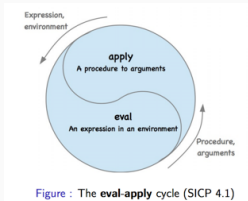
结束

- 好，分享结束，大家可以回去看书了。
- 预计一年后，应该可以成功写出来了。
- 开个玩笑，我们继续。看一个 半小时就能写出来的版本 。

半小时版本

核心逻辑

- parse \rightarrow (eval¹ \rightarrow apply²) loop...



¹处理表达式

²处理值

解析 (1 min)

这里我们就偷个懒，利用 `lisp` 的 `read-from-string` / `read` 方法

```
1 (read-from-string "(1 2 3)")
2   ;;; => (1 2 3)
3 (read-from-string "1")
4   ;;; => 1
5 (read-from-string "nil")
6   ;;; => NIL
7 (read-from-string "(defun plus (a b ) (+ a b))")
8   ;;; => (DEFUN PLUS (A B) (+ A B))
9 (read-from-string "(defun plus (a b)")
10  ;;; Exception
```


表达式类型

- 原子
 - 常量 1, "abc"
 - 变量 a, test
- 列表
 - quote (quote (a b c))
 - if (if t 1 2)
 - lambda (lambda (a) (+ 1 a))
 - define (define a 1)
 - assign (set! a 2)
 - begin (begin (define b 1) (set! b 2))
 - apply procedure (plus 1 2)

原子表达式

符号

当遇到了一个符号的时候，从当前的上下文中去查找其对应的值，做替换

```
1 a ;;; nil or unbound exception  
2 (set! a 1) ;;; => 1  
3 a ;;; => 1
```

常量

常量表达式的值即为本身

```
1 1 ;;; => 1  
2 "abc" ;;; "abc"
```

特殊形式 `if`

```
1 (if predicate consquence alternative)
```

先求值 `predicate`，如果符合，则求值 `consquence`，反之，则求值 `alternative`

特殊在于，控制表内的求值顺序。

并不会将表内表达式均求值，而是根据第一个元素的值，来决定后续如何进行求值。

特殊形式 `define` 以及 `set!`

```
1 (define variable value)
2 (set! variable value)
```

只求值 `value`, 并将求值后的结果赋值给 `variable`³

特殊在于, 控制表内的求值顺序以及 修改上下文
不对 `variable` 求值, 仅求值 `value`, 而后修改上下文。

³赋值表示在上下文中添加 (`install`) 这个符号以及这个符号对应的值.

特殊形式 `quote`

```
1 (quote (a b c))
```

返回其引用的表达式

syntactic sugar: `'(a b c)`

特殊在于, 控制表内的求值顺序。

不对表达式内求值, 仅返回其引用的表达式。

特殊形式 `lambda`

```
1 (lambda (a) (+ 1 a))
```

生成一个 `procedure`, 包含 **形式参数** 列表, 以及 **待执行的表达式** 列表。

特殊同样在于, 控制表内的求值顺序。

只将待执行的表达式记录下来, 留待需要时使用。

特殊形式 `begin`

```
1 (begin  
2   (define a 3)  
3   (set! a 1)  
4   (+ a 2))
```

依次执行表达式序列

特殊在于, 控制表内的求值顺序

函数调用 i

```
1 (define plus (lambda (a) (+ 1 a)))  
2 (plus 2) ;;; 3
```

1. 求值操作符

2. 求值操作数

3. 应用该操作

- 系统方法，则直接调用下层的 *apply*
- 自定义的方法

3.1 把形参对应的值添加到上下文中，生成新的上下文

3.2 在新的上下文中，求值表达式列表

函数定义

```
1  ;;; env => ((+) (<primitive +>))
2  (define plus1 (lambda (a) (+ 1 a)))
3  ;;; env =>
4  ;;;      ( (+ plus1)
5  ;;;      (<primitive +> <procedure object>) )
6
7  (plus1 2)
```

执行过程

```
1 (eval 'plus1 env) ;;; <procedure object>
2 (eval 2) ;;; 2
3 (extend (a) (2) env)
4 ;;; env' => ( (+ plus1 a)
5 ;;;          (<primitive +> <procedure object> 2))
6
7 (eval '(+ 1 a) env')
8 (eval a env') ;;; 2
9 (apply-primitive '+ (1 2)) ;;; 3
```

求值环境/上下文

```
(define make-account  
  (lambda (balance)  
    (lambda (amt)  
      (begin (set! balance (+ balance amt))  
              balance)))))
```

```
(define account1 (make-account 100.00))  
(account1 -20.00)
```

```
+: <built-in operator add>  
make-account: <a Procedure>
```

```
balance: 100.00  
amt: -20.00
```

```
account1: <a Procedure>
```

实现一个解释器

声明

源代码来自 SICP 第 4 章，链接见附录。

eval (dispatch)

```
1 (define (eval exp env)
2   (cond ((self-evaluating? exp) exp)
3         ((variable? exp) (lookup-variable-value exp ←
4           env))
5         ((quoted? exp) (text-of-quotation exp))
6         ((assignment? exp) (eval-assignment exp env))
7         ((definition? exp) (eval-definition exp env))
8         ((if? exp) (eval-if exp env))
9         ((lambda? exp)
10          (make-procedure (lambda-parameters exp)
11                           (lambda-body exp)
12                           env))
13         ((begin? exp)
14          (eval-sequence (begin-actions exp) env))
15         ((application? exp)
16          (apply (eval (operator exp) env)
17                  (list-of-values (operands exp) env)))
18         (else
19          (error "Unknown expression type — EVAL" exp)←
20          )))
```

apply

```
1 (define (apply procedure arguments)
2   (cond ((primitive-procedure? procedure)
3         (apply-primitive-procedure procedure ←
4           arguments))
5         ((compound-procedure? procedure)
6           (eval-sequence
7             (procedure-body procedure)
8             (extend-environment
9               (procedure-parameters procedure)
10              arguments
11              (procedure-environment procedure))))
12         (else
13           (error "Unknown procedure type — APPLY" procedure)←
14             )))
```

env 求值上下文 i

```
1 (define (enclosing-environment env) (cdr env))
2 (define (first-frame env) (car env))
3 (define the-empty-environment '())
4
5 (define (make-frame variables values)
6   (cons variables values))
7 (define (frame-variables frame) (car frame))
8 (define (frame-values frame) (cdr frame))
9 (define (add-binding-to-frame! var val frame)
10   (set-car! frame (cons var (car frame))))
11   (set-cdr! frame (cons val (cdr frame))))
```


env 求值上下文 ii

```
1 (define (extend-environment vars vals base-env)
2   (if (= (length vars) (length vals))
3       (cons (make-frame vars vals) base-env)
4       (if (< (length vars) (length vals))
5           (error "Too many arguments supplied" vars ↵
6               vals)
7           (error "Too few arguments supplied" vars ↵
8               vals))))
```

eval-atom i

```
1 (define (self-evaluating? exp)
2   (cond ((number? exp) true)
3         ((string? exp) true)
4         (else false)))
5
6 (define (variable? exp) (symbol? exp))
```

eval-atom ii

```
1 (define (lookup-variable-value var env)
2   (define (env-loop env)
3     (define (scan vars vals)
4       (cond ((null? vars)
5              (env-loop (enclosing-environment env)))
6             ((eq? var (car vars))
7              (car vals))
8             (else (scan (cdr vars) (cdr vals)))))
9     (if (eq? env the-empty-environment)
10        (error "Unbound variable" var)
11        (let ((frame (first-frame env)))
12          (scan (frame-variables frame)
13                (frame-values frame)))))
14   (env-loop env))
```

eval-define & eval-assign i

```
1 (define (eval-assignment exp env)
2   (set-variable-value!
3     (assignment-variable exp)
4     (eval (assignment-value exp) env)
5     env)
6   'ok)
7
8 (define (eval-definition exp env)
9   (define-variable!
10    (definition-variable exp)
11    (eval (definition-value exp) env)
12    env)
13   'ok)
```

eval-define & eval-assign ii

```
1 (define (define-variable! var val env)
2   (let ((frame (first-frame env)))
3     (define (scan vars vals)
4       (cond ((null? vars)
5              (add-binding-to-frame! var val frame))
6             ((eq? var (car vars))
7              (set-car! vals val))
8             (else (scan (cdr vars) (cdr vals)))))
9   (scan (frame-variables frame)
10         (frame-values frame)))
```

eval-if

```
1 (define (eval-if exp env)
2   (if (true? (eval (if-predicate exp) env))
3       (eval (if-consequent exp) env)
4       (eval (if-alternative exp) env)))
```

eval-quote

```
1 (define (quoted? exp)
2   (tagged-list? exp 'quote))
3
4 (define (text-of-quotation exp) (cadr exp))
```

eval-lambda

```
1 (define (make-procedure parameters body env)
2   (list 'procedure parameters body env))
3 (define (compound-procedure? p)
4   (tagged-list? p 'procedure))
5 (define (procedure-parameters p) (cadr p))
6 (define (procedure-body p) (caddr p))
7 (define (procedure-environment p) (cadddr p))
```


eval-begin

```
1 (define (eval-sequence exps env)
2   (cond ((last-exp? exps) (eval (first-exp exps) env))
3         (else (eval (first-exp exps) env)
4                   (eval-sequence (rest-exps exps) env))))
```

最后

几个题外话

- 如果没有 `assign`, 会不会简单很多
- 如果使用 `lazy` 的求值, 而不是应用时求值, 是否很多特殊形式就没有必要了
- 如果增加一个 `case` 的关键字
- 如果做语法分析
- 如果要编译成 `c`

- (How to Write a (Lisp) Interpreter (in Python))
- (An ((Even Better) Lisp) Interpreter (in Python))
- SICP Chapter 4: The Metacircular Evaluator