Latest: HOWTO: Get tenure
Next: 3 shell scripts that can improve your writing
Prev: Console productivity hack: Exploiting task frequency
Rand: Self-inlining anonymous functions in C++

7 lines of code, 3 minutes: Implement a programming language from scratch

[article index] [email me] [@mattmight] [+mattmight] [rss]

Implementing a programming language is an experience no programmer should go without; the process fosters a deep understanding of computation, and it's fun!

In this article, I've boiled the entire process down to its essence: a 7-line interpreter for a functional (Turing-equivalent) programming language. It takes about 3 minutes to implement.

This 7-line interpreter showcases a scalable architecture found in many interpreters--the eval/apply design pattern of Structure and Interpretation of Computer Programs:

In total, there are three language implementations in this article:

- a 7-line, 3-minute interpreter in Scheme;
- a re-implementation in Racket; and
- a 100-line, "1-afternoon" interpreter that implements top-level binding forms, explicit recursion, side effects, higher-order functions and more!

The last interpreter is a good starting point for growing a richer language.

A small (yet Turing-equivalent) language

The easiest programming language to implement is a minimalist, higher-order functional programming language known as the lambda calculus.

The lambda calculus actually lives at the core of all the major functional languages--Haskell, Scheme and ML--but it also lives inside JavaScript, Python and Ruby. It's even hiding inside Java, if you know where to find it.

A brief history

Alonzo Church developed the lambda calculus in 1929.

Back then, it wasn't called a programming language because there were no computers; there wasn't anything to "program."

It was really just a mathematical notation for reasoning about functions.

Fortunately, Alonzo Church had a Ph.D. student named Alan Turing.

Alan Turing defined the Turing machine, which became the first accepted definition of a general-purpose computer.

It was soon discovered that the lambda calculus and the Turing machine were equivalent: any function you could describe with the lambda calculus could be implemented on a Turing machine, and any function you could implement on a Turing machine could be described in the lambda calculus.

What makes this remarkable is that there are only three kinds of expressions in the lambda calculus: variable references, anonymous functions and function calls.

Anonymous functions

Anonymous functions are written with a "lambda-dot" notation, so that:

```
(λ v . e)
```

is the function that accepts an argument *v* and returns the value of *e*.

If you've programmed in JavaScript, this form is equivalent to:

```
function (v) { return e ; }
```

Function call

Function call is written by making two expressions adjacent:

```
(f e)
```

In JavaScript (or just about any other language), you'd write this as:

```
f(e)
```

Examples

The identity function, which just returns its argument, is easy to write:

```
(λ x . x)
```

We can apply the identity function to the identity function:

```
((λ x . x) (λ a . a))
```

(Which of course just returns the identity function.)

Here's a (slightly) more interesting program:

```
(((λ f . (λ x . (f x))) (λ a . a)) (λ b . b))
```

Can you figure out what it does?

Wait! How the heck is this a "programming" language?

At first glance, this simple language seems to lack both recursion and iteration, not to mention numbers, booleans, conditionals, data structures and all the rest. How can this language possibly be general-purpose?

The way that the lambda calculus achieves Turing-equivalence is through two of the coolest programming hacks out there: Church encodings and the Y combinator.

I've written an entire article on the Y combinator and another one on Church encodings. But, if you don't want to read those, I can convince you that there's more to the lambda calculus than you probably expected with just one program:

```
((λ f . (f f)) (λ f . (f f)))
```

This outwardly benign program is called Omega, and if you try to execute it, it doesn't terminate! (See if you can figure out why.)

Implementing the lambda calculus

Below is the 7-line, 3-minute interpreter for the lambda calculus, in R5RS Scheme. In technical terms (explained below), it's an environment-based denotational interpreter.

```
; eval takes an expression and an environment to a value
(define (eval e env) (cond
  ((symbol? e)       (cadr (assq e env)))
  ((eq? (car e) 'λ)  (cons e env))
  (else              (apply (eval (car e) env) (eval (cadr e) env)))))

; apply takes a function and an argument to a value
(define (apply f x)
  (eval (cddr (car f)) (cons (list (cadr (car f)) x) (cdr f))))

; read and parse stdin, then evaluate:
(display (eval (read) '())) (newline)
```

This code will read a program from stdin, parse it, evaluate it and print the result. (It's 7 lines without the comments and blank lines.)

Scheme's `read` function makes lexing and parsing simple--as long as you're willing to live in the "balanced parentheses" (i.e. s-expression) world of syntax. (If not, you'll have to read up on lexing in parsing; you can start with one of my articles on lexing.) In Scheme, `read` grabs properly parenthesized input from stdin and parses it into a tree.

Two functions form the core of the interpreter: `eval` and `apply`. Even though we're in Scheme, we can give conceptual "signatures" to these functions:

```
eval  : Expression * Environment -> Value
apply : Value * Value -> Value

Environment = Variable -> Value
Value       = Closure
Closure     = Lambda * Environment
```

The `eval` function takes an expression and an enviroment to a value. The expression can be either a variable, a lambda term or an application.

An environment is a map from variables to values, used to define the free variables of an open term. (An open term is one that has unbound occurrences of a variable.) Consider, for example, the expression `(λ x . z)`. This term is open, because we don't know what `z` is.

Since we're in R5RS Scheme, we can use associative lists to define environments.

A closure is an encoding of a function that pairs a (possibly open) lambda expression with an environment to define its free variables. In other words, a closure closes an open term.

A cleaner implementation in Racket

Racket is a batteries-included, get-stuff-done dialect of Scheme. Racket provides a match construct that cleans up the interpreter. Check it out:

```
#lang racket

; bring in the match library:
(require racket/match)

; eval matches on the type of expression:
(define (eval exp env) (match exp
  [`(,f ,e)        (apply (eval f env) (eval e env))]
  [`(λ ,v . ,e)   `(closure ,exp ,env)]
  [(? symbol?)     (cadr (assq exp env))]))

; apply destructures the function with a match too:
(define (apply f x) (match f
  [`(closure (λ ,v . ,body) ,env)
    (eval body (cons `(,v ,x) env))]))

; read in, parse and evaluate:
(display (eval (read) '()))    (newline)
```

This one is larger, but it's cleaner and easier to understand.

A bigger language

The lambda calculus is a tiny language. Even so, the eval/apply design of its interpreter scales to much bigger languages. For instance, in about 100 lines, we can implement an interpreter for a sizeable subset of Scheme itself.

Consider a language with an assortment of expression forms:

1. Variable reference; ex: `x`, `foo`, `save-file`.
2. Numeric and boolean constants; ex: `300`, `3.14`, `#f`.
3. Primitive operations; ex: `+`, `-`, `<=`.
4. Conditionals: `(if` *`condition if-true if-false`*`)`.
5. Variable bindings: `(let ((`*`var value`*`) ...)` *`body-expr`*`)`.
6. Recursive bindings: `(letrec ((`*`var value`*`) ...)` *`body-expr`*`)`.
7. Variable mutation: `(set!` *`var value`*`)`.
8. Sequencing: `(begin` *`do-this then-this`*`)`.

Now add three top-level forms to this language:

1. Function definition: `(define (`*`proc-name var ...`*`)` *`expr`*`)`.
2. Global definition: `(define` *`var expr`*`)`.
3. Top-level expression: *`expr`*.

Here's the entire interpreter, with test harness and test cases included:

```
#lang racket

(require racket/match)

;; Evaluation toggles between eval and apply.

; eval dispatches on the type of expression:
(define (eval exp env)
  (match exp
    [(? symbol?)          (env-lookup env exp)]
    [(? number?)          exp]
    [(? boolean?)         exp]
    [`(if ,ec ,et ,ef)    (if (eval ec env)
                              (eval et env)
                              (eval ef env))]
    [`(letrec ,binds ,eb) (eval-letrec binds eb env)]
    [`(let    ,binds ,eb) (eval-let binds eb env)]
    [`(lambda ,vs ,e)     `(closure ,exp ,env)]
    [`(set! ,v ,e)        (env-set! env v e)]
    [`(begin ,e1 ,e2)     (begin (eval e1 env)
                                 (eval e2 env))]
    [`(,f . ,args)        (apply-proc
                           (eval f env)
                           (map (eval-with env) args))]))

; a handy wrapper for Currying eval:
(define (eval-with env)
  (lambda (exp) (eval exp env)))
```

```
; eval for letrec:
(define (eval-letrec bindings body env)
  (let* ((vars (map car bindings))
         (exps (map cadr bindings))
         (fs   (map (lambda _ #f) bindings))
         (env* (env-extend* env vars fs))
         (vals (map (eval-with env*) exps)))
    (env-set!* env* vars vals)
    (eval body env*)))

; eval for let:
(define (eval-let bindings body env)
  (let* ((vars (map car bindings))
         (exps (map cadr bindings))
         (vals (map (eval-with env) exps))
         (env* (env-extend* env vars vals)))
    (eval body env*)))

; applies a procedure to arguments:
(define (apply-proc f values)
  (match f
    [`(closure (lambda ,vs ,body) ,env)
     ; =>
     (eval body (env-extend* env vs values))]

    [`(primitive ,p)
     ; =>
     (apply p values)]))

;; Environments map variables to mutable cells
;; containing values.

(define-struct cell ([value #:mutable]))

; empty environment:
(define (env-empty)  (hash))

; initial environment, with bindings for primitives:
(define (env-initial)
  (env-extend*
   (env-empty)
   '(+  -  /  *  <=  void  display  newline)
   (map (lambda (s) (list 'primitive s))
   `(,+ ,- ,/ ,* ,<= ,void ,display ,newline))))

; looks up a value:
(define (env-lookup env var)
  (cell-value (hash-ref env var)))

; sets a value in an environment:
(define (env-set! env var value)
  (set-cell-value! (hash-ref env var) value))

; extends an environment with several bindings:
(define (env-extend* env vars values)
  (match `(,vars ,values)
    [`((,v . ,vars) (,val . ,values))
     ; =>
     (env-extend* (hash-set env v (make-cell val)) vars values)]

    [`(() ())
     ; =>
     env]))

; mutates an environment with several assignments:
```

```
(define (env-set!* env vars values)
  (match `(,vars ,values)
    [`((,v . ,vars) (,val . ,values))
     ; =>
     (begin
       (env-set! env v val)
       (env-set!* env vars values))]

    [`(() ())
     ; =>
     (void)]))

;; Evaluation tests.

; define new syntax to make tests look prettier:
(define-syntax
  test-eval
  (syntax-rules (====)
    [(_ program ==== value)
     (let ((result (eval (quote program) (env-initial))))
       (when (not (equal? program value))
         (error "test failed!")))]))

(test-eval
  ((lambda (x) (+ 3 4)) 20)
  ====
  7)

(test-eval
  (letrec ((f (lambda (n)
                (if (<= n 1)
                    1
                    (* n (f (- n 1)))))))
    (f 5))
  ====
  120)

(test-eval
  (let ((x 100))
    (begin
      (set! x 20)
      x))
  ====
  20)

(test-eval
  (let ((x 1000))
    (begin (let ((x 10))
             20)
           x))
  ====
  1000)

;; Programs are translated into a single letrec expression.

(define (define->binding define)
  (match define
    [`(define (,f . ,formals) ,body)
     ; =>
     `(,f (lambda ,formals ,body))]

    [`(define ,v ,value)
     ; =>
     `(,v ,value)]
```

```
    [else
     ; =>
     `(,(gensym) ,define)]))

(define (transform-top-level defines)
  `(letrec ,(map define->binding defines)
     (void)))

(define (eval-program program)
  (eval (transform-top-level program) (env-initial)))

(define (read-all)
  (let ((next (read)))
    (if (eof-object? next)
        '()
        (cons next (read-all)))))

; read in a program, and evaluate:
(eval-program (read-all))
```

You can download the source: minilang.rkt.

From here

You should be able to quickly test out new ideas for programming languages by modifying the last interpreter.

If you want a language with different syntax, you can build a parser that dumps out s-expressions. Taking this approach cleanly separates syntactic design from semantic design.

More resources

- Until recently, MIT taught introductory computer science by having students build interpreters. The textbook for that class, Structure and Interpretation of Computer Programs, is a modern classic.
- Lisp in Small Pieces is a well-written tome on implementing (Scheme-like) functional programming languages. It's probably going to be the "textbook" for my upcoming scripting language design and implementation class.
- My articles on the Y combinator and Church encodings.
- My article on implementing first-class macros. The interpreter uses the same eval/apply architecture.

Related posts

- Tree transformations: Desugaring Scheme
- Lexical analysis in Racket
- Grammar: The language of languages (BNF, EBNF, ABNF)

- What is static program analysis?
- Implementing Java as a CESK machine, in Java
- Writing an interpreter, CESK-style
- Order theory for computer scientists
- HOWTO: Translate math into code
- Writing CEK-style interpreters in Haskell
- Closure conversion: How to compile lambda
- How to compile with continuations
- Understand exceptions by implementing them
- A-Normalization: Why and How
- Compiling up to the λ-calculus
- Parsing with derivatives (Yacc is dead: An update)
- By example: Continuation-passing style in JavaScript
- Architectures for interpreters
- First-class macros from meta-circular evaluators
- Programming with continuations by example
- Compiling Scheme to C
- Compiling to Java
- Church encodings in Scheme
- Non-termination without loops, iteration or recursion in Javascript
- Memoizing recursive functions in Javascript with the Y combinator
- Advanced programming languages
- Recommended books and papers for grad students

[article index] [email me] [@mattmight] [+mattmight] [rss]

Latest: HOWTO: Get tenure
Next: 3 shell scripts that can improve your writing
Prev: Console productivity hack: Exploiting task frequency
Rand: Self-inlining anonymous functions in C++