As Raízes de Lisp

Paul Graham
Tradução: Tiago Charters*

21 de Abril de 2009

John McCarthy em 1960 publicou um artigo espantoso no qual fez para a programação o mesmo que Euclides fez para a geometria[1]. Mostrou como construir, dado uma mão cheia de operadores simples e uma notação para funções, uma linguagem de programação completa. Denominou-a de *Lisp*, abreviatura de "*List Processing*", porque a ideia chave era usar uma estrutura simples de dados chamada **lista** para o código e para os dados propriamente ditos.

O que McCarthy descobriu não foi só aquilo que se tornou um marco para a história dos computadores, e que vale a pena perceber por si só, mas um modelo para o qual converge a programação neste tempo. Parece-me a mim que existem apenas dois tipos de modelos realmente escorreitos e consistentes: o modelo do C e o de Lisp. Estes dois modelos parecem estar bem alto com baixios pantanosos entre si. Com o aumento da capacidade dos computadores as linguagens de programação que têm surgido convergem para o modelo de Lisp. A receita mais popular para construir uma nova linguagem de programação nos últimos 20 anos consiste em considerar o modelo de C e adicionar partes do modelo de Lisp, tais como *runtime typing* e *garbage collection*.

Neste artigo tentarei explicar de uma maneira simples o que McCarthy descobriu. O objectivo não é só o de aprender um resultado teórico importante que alguém descobriu à quarenta anos atrás, mas mostrar para onde tendem as linguagens de programação. A coisa pouco usual do Lisp – de facto, a qualidade que o define – é que se escreve a si mesmo. Para se entender o que McCarthy quer dizer com isto, vamos repetir os seus passos, traduzindo a sua notação matemática em código Common Lisp.

*tca@cii.fc.ul.pt

[1] "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications of the ACM 3:4, April 1960, pp. 184-195.*

1 Sete operadores primitivos

Para começar, definimos o que se entende por **expressão**. Uma expressão é, alternativamente, um **átomo**, que é uma sequência de letras (por exemplo: `foo`), ou uma **lista** de zero ou mais expressões, separadas por espaços em branco e enquadradas por parêntesis. De seguida mostram-se várias expressões:

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

A última expressão é uma lista de quatro elementos, o terceiro dos quais é uma lista de um elemento.

Em aritmética a expressão $1+1$ tem o valor 2. Uma expressão válida de Lisp também tem um valor. Se uma expressão e tem o valor v dizemos que *e return v*. O passo seguinte é definir que tipo de expressões podem existir e que tipo de valor têm.

Se uma expressão é uma lista, chamamos ao primeiro elemento da lista **operador**, e os restantes elementos **argumentos**. Vamos, no que se segue, definir sete operadores primitivos (no sentido de axiomas): `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, e `cond`.

1. `(quote` x`)` retorna x. Para uma leitura mais simples abreviamos `(quote` x`)` por `'`x.

   ```
   > (quote a)
   a
   > 'a
   a
   > (quote (a b c))
   (a b c)
   ```

2. `(atom` x`)` retorna o átomo `t` se o valor de x é um átomo ou a lista vazia `()`. Em Lisp convenciona-se usar o átomo `t` como representando o valor lógico verdade, e a lista vazia como o valor lógico falso.

   ```
   > (atom 'a)
   t
   > (atom '(a b c)
   ()
   > (atom '())
   t
   ```

Agora que temos um operador cujo argumento é avaliado podemos mostrar para que é que `quote` serve. Através de *quoting*[2] uma lista protegemo-la de ser avaliada. Uma lista *unquoted* como argumento de um operador como por exemplo `atom` é tratada como código:

```
> (atom (atom ' a))
> t
```

enquanto uma lista *quoted* é tratada como uma mera lista, neste caso uma lista de dois elementos

```
> (atom '(atom 'a))
()
```

Isto corresponde à maneira de usar citações em Inglês[3]. Cambridge é uma cidade de Massachusetts que contém cerca de 90.000 habitantes. "Cambridge" é uma palavra de nove letras.

`quote` pode parecer um conceito estranho, não há muitas línguas com coisa semelhante. Está intimamente ligado a uma das características distintivas de Lisp: o código e os dados são feitos de uma mesma estrutura, e o operador *quote* é uma maneira de distinguir os dois.

3. `(eq` x y`)` retorna `t` se os valores de x e y são o mesmo átomo ou ambos uma lista vazia, e `()` caso contrário.

```
> (eq 'a 'a)
t
> (eq 'a 'b)
()
> (eq '() '())
t
```

4. `(car` x `)` espera que o valor de x seja uma lista, e retorna o seu primeiro elemento.

```
> (car '(a b c))
a
```

5. `(cdr` x `)` espera que o valor de x seja uma lista, e retorna o resto da lista a seguir do primeiro elemento.

[2]Nota do tradutor: não vou traduzir "quote" por citação de modo a que não se perca o sentido em Inglês da frase.

[3]Nota do tradutor: ver o que se passa em PT.

```
> (cdr '(a b c))
(b c)
```

6. `(cons` x y `)` espera que o valor de y seja uma lista, e retorna uma lista contendo o valor de x seguido dos elementos valor de y.

```
> (cons 'a '(b c))
(a b c)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (car (cons 'a '(b c)))
a
> (cdr (cons 'a '(b c)))
(b c)
```

7. `(cond` $(p_1\ e_1)\ \ldots\ (p_n\ e_n)$ `)` e avaliado da seguinte forma. As expressões p são avaliadas até que uma retorna `t`. Quando uma é encontrada, o valor correspondente da expressão e é retornado como o valor total da expressão `cond`.

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a) 'second))
second
```

Os argumentos de cinco dos sete operadores primitivos anteriores são avaliados quando uma expressão com esse operador é avaliado. Chamaremos a esses operadores *funções*.

2 Denotando funções

De seguida definimos a notação para a descrição de funções. Uma função é representada por `((lambda (`$p_1\ \ldots\ p_n$`) `e`)` $a_1\ \ldots a_n$`)`, onde $p_1\ \ldots\ p_n$ são átomos (chamados **parâmetros**) e e é uma expressão. Uma expressão cujo primeiro elemento é uma expressão

`((lambda (`$p_1\ \ldots\ p_n$`) `e`)`$a_1\ \ldots a_n$`)`

é chamada de **função** e o seu valor é calculado da seguinte forma. Cada expressão a_i é avaliada. Depois e é avaliada. Durante a avaliação de e, o valor de qualquer ocorrência de qualquer um dos p_i é o valor do a_i correspondente na mais recente chamada da função.

```
> ((lambda (x) (cons x '(b))) 'a)
(a b)
> ((lambda (x y) (cons x (cdr y))) 'z '(a b c))
(z b c)
```

Se uma expressão tem como primeiro elemento um átomo f que não é um dos operadores primitivos

(*f* a_1 ... a_n)

e o valor de f é uma função (lambda (p_1 ... p_n) e) então o valor da expressão é o valor de

((lambda(p_1 ... p_n) e) $a_1 \ldots a_n$)

Por outras palavras, os parâmetros podem ser usados com expressões assim como argumentos[4]:

```
> ((lambda (f) (f '(b c)))
   '(lambda (x) (cons 'a x)))
```

Existe outra notação para funções que permite que uma função se chame a si própria, permitindo assim de uma forma conveniente definir funções recursivas (definidas por recorrência)[5].

A notação label f (lambda (p_1 ... p_n) e) denota a função que se comporta com lambda (p_1 ... p_n) e), com a propriedade adicional que uma ocorrência de f dentro de e avaliará a expressões label, como se f fosse um parâmetro da função.

Suponha-se que queremos construir uma função subst(x y z), que toma uma expressão x, um átomo y e uma lista z, e retorna uma lista tipo z onde cada instância y (em qualquer nível) é substituída em z por x.

```
> (subst 'm ' b '(a b (a b c) d))
(a m (a m c) d)
```

Podemos escrever esta função como

```
(label subst (lambda (x y z)
               (cond ((atom z)
                      (cond (eq z y) x)
                            ('t z))
                     ('t (cons (subst x y (car z))
                               (subst x y (cdr z)))))))
```

Vamos abreviar f = (label f (lambda (p_1 ... p_n) e)) por (defun f (p_1 ... p_ne)) e logo

```
(defun subst (x y z)
  (cond ((atom z)
         (cond ((eq z y) x)
```

[4]Não funciona o exemplo!

[5]Não é necessário definir uma nova notação para isto. Poderia-mos definir funções recursivas com nossa notação através de do construtor Y. McCarthy poderia não conhecer o Y construtor na altura em que escreveu o artigo; de qualquer modo, a notação label é de leitura mais simples

```
        ('t z)))
  ('t (cons (subst x y (car z))
            (subst x y (cdr z))))))
```

Acidentalmente, podemos ver que uma condição definida por omissão numa expressão `cond`. Uma condição cujo primeiro elemento é `'t` é sempre executada. Assim `(cond(` x y`)('t` z`))` é equivalente aquilo que se poderia escrever numa outra linguagem com sintaxe

`if` x `then` y `else` z

3 Algumas funções

Agora que temos uma maneira de expressar funções, podemos definir novas em termos dos nossos sete operadores primitivos. Em primeiro lugar é conveniente introduzir algumas abreviaturas. Vamos usar `c`x`r`, como uma sequência de `a`s ou `d`s, correspondendo a composição de `car` e `cdr`. Por exemplo `(cadr`e`)` é uma abreviatura de `(car (cdr` e`))`, que retorna o segundo elemento de e.

```
> (cadr '((a b) (c d) e))
(c d)
> (caddr '((a b) (c d) e))
e
> (cdar '((a b) (c d) e))
(b)
```

Vamos também usar `list` $e_1 \ldots e_n$`)` para significar `(cons` e_1 `...(cons` e_n`'())...)`.

```
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> ( list 'a 'b 'c)
(a b c)
```

Vamos definir novas funções. Alterei os nomes das funções que se seguem introduzindo um ponto o fim. Isto distingue as funções primitivas daquelas definidas à custa delas, e também evita confusões com aquelas que já existem em Commnon Lisp.

1. `(null.` x`)` testa se o seu argumento é uma lista vazia.

   ```
   (defun null. (x)
     (eq x '()))

   > (null. 'a)
   ()
   > (null. '())
   t
   ```

2. (`and.` x y) retorna `t` se ambos os argumentos o são, `()` caso contrário.

```
(defun and. (x y)
  (cond (x (cond (y't) ('t '())))
        ('t '())))

> (and. (atom 'a) (eq 'a 'a))
t
>  (and. (atom 'a) (eq 'a 'b))
()
```

3. (`not.` x) retorna `t` se o seu argumento é `()`, e `()` se o seu argumento é `t`.

```
(defun not. (x)
  (cond (x '())
        ('t 't)))

> (not. (eq 'a 'a))
()
>  (not. (eq 'a 'b))
t
```

4. (`append.` x y) toma duas listas e retorna a sua concatenação.

```
(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

> (append.'(a b) '(c d))
(a b c d)
> (append. '() '(c d))
(c d)
```

5. (`pair.` x y) toma duas listas com o mesmo comprimento e retorna uma lista de listas de dois elementos, com pares de formados com elementos de cada uma das listas iniciais.

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list (car x) (car y))
               (pair. (cdr x) (cdr y))))))

> (pair. '(x y z) '(a b c))
((x a) (y b) (z c))
```

6. `(assoc.` x y`)` toma um átomo x e uma lista y da forma criada por `pair.`, e retorna o segundo elemento da primeira lista de y que contém x como primeiro elemento.

```
(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))

> (assoc. 'x '((x a) (y b)))
a
> (assoc. 'x '((x new) (x a) (y b)))
new
```

4 A surpresa

Assim conseguimos definir funções que concatenam listas, substituem uma expressão por outra, etc. Talvez seja apenas uma notação elegante, e depois? Agora chega a surpresa. Podemos também escrever uma função que funciona como um interpretador da nossa linguagem: uma função que tem como argumento uma expressão de Lisp, retornando o seu valor. Aqui está ela:

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) ' car)  (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
```

```
                    a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval. (car m) a)
            (evlis. (cdr m) a)))))
```

A definição de `eval.` é bastante maior do que as definições que vimos anteriormente. Vamos estuda-la em mais detalhe e por partes.

A função tem dois argumentos: `e`, a expressão a ser avaliada,e `a`, uma lista que representa os valores que os átomos que aparecem como parâmetros das chamadas das funções. Esta última lista é chamada de **ambiente**, e é da forma da listas criadas por `pair.` e `assoc.`.

A parte principal da função `eval.` é a expressão `cond.` que tem quatro condições (clausulas). A avaliação de uma expressão depende de qual é usada. A primeira clausula trata de átomos. Se `e` é um átomo, procuramos o seu valor no ambiente:

```
> (eval. 'x '((x a) (y b)))
a
```

A segunda clausula de `eval.` é um outro `cond` que trata de expressões da forma $(a\ \ldots)$, onde a é um átomo. Estas incluem todos os tipos de operadores primitivos, e há uma clausula para cada um deles.

```
> (eval. '(eq 'a 'a) '())
t
> (eval. '(cons x '(b c))
         '((x a ) (y b)))
(a b c)
```

Todos eles (excepto `quote`) chamam `eval.` para determinar o valor dos seus argumentos.

As duas últimas clausulas são mais complicadas. Para avaliar uma expressão `cond` chamamos uma função subsidiária `evcon.`, que percorre as clausulas de uma forma recursiva, procurando o primeiro elemento que retorna `t`. Quando o encontra tal clausula retorna o valor do segundo elemento.

```
> (eval. '(cond ((atom x) 'atom)
                ('t 'list))
         '((x '(a b))))
list
```

A parte final da segunda clausula de `eval.` trata das chamadas das funções que são passadas como parâmetros. Funciona substituindo o átomo pelos seu valor (que deverá ser uma expressão `lambda` ou `label`)) avaliando o resultado da expressão. Assim

```
(eval. '(f '(b c))
       '((f lambda (x) (cons 'a x))))
```

passa a ser

```
(eval. '((lambda (x) (cons 'a x)) '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

que retorna `(a b c)`.

As duas últimas clausulas de `eval.` tratam de chamadas de funções para as quais o primeiro elemento da lista é uma expressão `lambda` ou `label`. Uma função `label` é avaliada empurrando o nome da função e a própria função para o ambiente, e depois chamando `eval.` numa expressão onde a expressão `lambda` interior é substituída pela expressão `label`. Isto é,

```
(eval. '((label firstatom (lambda (x)
                            (cond ((atom x) x)
                                  ('t (firstatom (car x)))))))
           y)
[...]
```

que no fim retorna `a`.

Finalmente, uma expressão da forma `((lambda (`$p_1 \ldots p_n$`) e)` $a_1 \ldots a_n$`)`, é avaliada chamando em primeiro lugar `evlis.` de modo a obter uma lista de valores `(` $v_1 \ldots v_n$`)` dos argumentos `(` $a_1 \ldots a_n$`)`, avaliando e com `(`$p_1 v_1$`)` $\ldots$ `(`$p_n v_n$`)` "appended" ao ambiente. Assim

```
[...]
```

fica

```
[...]
```

que no fim retorna `(a c d)`.

5 *Aftermath*

Agora que percebemos como é que `eval` funciona, voltemos atrás para perceber qual é o seu significado. O que aqui temos é um modelo muito elegante para computação. Usando apenas `quote`, `atom`, `car`, `cdr`, `cons` e `cond`, podemos definir a função `eval.`, que implementa a nossa linguagem de programação, usando-a depois para construir qualquer outra função que queiramos.

Como é claro, na altura já existiam modelos de computação – o mais notável exemplo sendo as máquinas de Turing. Mas os programas das máquinas de Turing não são muito edificantes de se ler. Se queremos uma linguagem para descrever algoritmos precisamos de algo mais abstracto, este foi um dos objectivos de McCarthy ao definir Lisp.

A linguagem definida em 1960 tinha muitas omissões. Não tinhas *side-effects*, nem execução sequencial (que apenas é útil com o anterior), sem números[6] e *dynamic scope*. Todas estas limitações podem ser remediadas, surpreendentemente, com um pouco de código adicional. Isto mesmo foi mostrado por Steele e Sussman num famoso artigo intitulado "*The Art of the Interpreter*"[7]

O entendimento do `eval` de McCarthy não é só perceber um patamar da história das linguagens. Estas ideias são ainda hoje parte do núcleo semântico do Lisp. Não é tanto o que McCarthy desenhou ou o que descobriu. Não é uma linguagem intrinsecamente para a IA ou para qualquer outra tarefa. É aquilo que obtemos quando tentamos axiomatizar a computação.

Com o passar do tempo, a linguagem média de programação (a linguagem usada pelo programador médio) tem-se aproximado do Lisp. Assim ao perceber `eval` estamos a perceber o que poderá ser o modelo principal de computação do futuro.

[6] É possível fazer cálculos de aritmética no Lisp de 1960 de McCarthy, usando, por exemplo, uma lista de n átomos para representar um número n.

[7] Guy Lewis Steele, Jr. and Gerald Jay Sussman, "The Art of the Interpreter, or the Modularity Complex (Part Zero, One and Two)", MIT AI Lab Memo 453, May 1978.

A O código

```
; The Lisp defined in McCarthy's 1960 paper, translated into CL.
; Assumes only quote, atom, eq, cons, car, cdr, cond.
; Bug reports to lispcode@paulgraham.com.

(defun null. (x)
  (eq x '()))

(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))

(defun not. (x)
  (cond (x '())
        ('t 't)))

(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

(defun list. (x y)
  (cons x (cons y '())))

(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list. (car x) (car y))
               (pair. (cdr x) (cdr y))))))

(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))

(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
```

```
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list. (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```