Construindo uma linguagem minimal

Quanto de redundância existe em uma linguagem? Qual o mínimo de contruções para se ter computação completa?

Ortogonalidade: uma única forma de escrever cada operação.
Compare: *awk* $\times$ *perl*.
Outras linugagens: *C*, *whitespace*

Scheme

Em *Scheme* o conjunto de contruções é relativamente pequeno:

- variáveis, símbolos, números, aritmética
- **define-type** e **type-case**
- **cons**, **first rest**
- **cond if**
- booleanos: **and or**
- chamada de função
- **local define lambda**

Muitos desses são construções a partir de outros (**car**, **cdr**, etc).
Qual o conjunto minimal de primitivas?

Listas

Sabemos que com listas e aplicação de funções podemos fazer quase qualquer coisa, como continuações, *environment*, estruturas de dados, *closures*....

Vamos começar definindo 4 grandes classes e tentar trabalhar com cada uma delas:

1. Variáveis, procedimentos e aplicação de funções
2. Números e aritmética
3. Valores e operações booleanas
4. Listas e estruras de dados agregados

Modelo

Como “caso modelo”, a função fatorial contém a maioria das características necessárias:

```
(define (fact n)
        (if (zero? n)
            1
            (* n (fact (sub1 n)))))
```

Listas

Listas parecem ser uma das estruturas mais básicas, mas pode ser mais simples ainda!

Pares: (pair A B)
é similar ao **cons**, bem geral.

Como construir o "pair"?
Deve receber 2 argumentos e retornar alguma coisa.
Chamada de procedimento é fundamental o suficiente para querermos manter$\rightarrow \lambda$!

$$(pair AB) \rightarrow (\lambda \cdots)$$

Construção de pares

Qual o argumento deste λ?
Outro λ, claro, que serve como seletor

$$(pair AB) \equiv (\lambda(sel)(sel AB))$$

Assim podemos escolher o primeiro ou o segundo:

$$\begin{aligned} left &\equiv (\lambda\,(A\,B) \quad A) \\ right &\equiv (\lambda\,(A\,B) \quad B) \end{aligned}$$

Chamada

Só precisamos inverter a chamada, pois o par é uma função. Se e é um par, chamamos:

$$(e \quad left)$$

$$(e \quad right)$$

Isto parece familiar???

Tudo pronto para começarmos...

Condicionais e booleanos

Um condicional (**if**) é um seletor entre dois valores.Ou seja, um *pair*:

$$(\mathbf{if}\, C\, T\, F) \equiv (C\, T\, F)$$

Verdadeiro e falso são facilmente implementados como condicionais (instâncias de um condicional):

$$\begin{aligned} yes &\equiv (\lambda\, (T\, F) \quad T) \\ no &\equiv (\lambda\, (T\, F) \quad N) \end{aligned}$$

Como fica a presença de *lazyness*, dá para retirar?
E **and** e **or**? Como fazer?

Números

Definição e ideia de número. Formalismo $\times$ intuicionismo.

O intuicionismo ajuda na abordagem cosntrutivista. Como aparecem os números?Estão associados a sequência e passagem de tempo.

A unidade, neste sentido, é contar uma vez, um ato isolado, uma única aplicação de função.Qual função?
Qualquer uma, não é a função que importa, mas sim o ato de aplicá-la.

$$um \equiv (\lambda\,(f)\,(\lambda\,(x)\,(f\;x)))$$

Enumeração

O resto vem por analogia:

$$
\begin{array}{rcl}
\mathit{zero} & \equiv & (\lambda\,(f)\,(\lambda\,(x)\,x)) \\
\mathit{um} & \equiv & (\lambda\,(f)\,(\lambda\,(x)\,(f\,x))) \\
\mathit{dois} & \equiv & (\lambda\,(f)\,(\lambda\,(x)\,(f\,(f\,x)))) \\
\mathit{três} & \equiv & (\lambda\,(f)\,(\lambda\,(x)\,(f\,(f,(f\,x))))) \\
\mathit{quatro} & \equiv & (\lambda\,(f)\,(\lambda\,(x)\,(f\,(f\,(f,(f\,x))))))
\end{array}
$$

E assim por diante. Dá até para definir o *zero*

Sucessor

Com a definição de sucessor, podemos começar a ter aritmética.

$$(succ\ um) \equiv (succ \quad (\lambda(f)\ (\lambda(x)\ (f\,x))))$$

deve valer *dois*:

$$(\lambda\,(f)\,(\lambda\,(x)\,(f\,(f\,x))))$$

Melhor:

$$(\lambda(f) \quad (\lambda(x) \quad (f\ ((um\ f)\ x))))$$

Ou seja, uma aplicação a mais.

..continuação

$$
\begin{aligned}
&(succ\ n) \equiv \\
&\qquad\qquad (\lambda\ (n) \\
&\qquad\qquad\quad (\lambda\ (f) \\
&\qquad\qquad\qquad (\lambda\ (x) \\
&\qquad\qquad\qquad\quad (f\ ((n\ f)\ x)))))
\end{aligned}
$$

É importante lembrar que n é uma função. Ou melhor, a aplicação sucessiva de uma função n vezes.

Soma e produto

A soma de m a n é simplesmente achar o m-ésimo sucessor de n.

$$
\begin{aligned}
&soma \equiv \\
&\qquad\quad (\lambda\ (m) \\
&\qquad\qquad (\lambda\ (n) \\
&\qquad\qquad\quad ((n\ succ)\ m)))
\end{aligned}
$$

O produto de m por n é repetir m vezes a n-ésima sucessão de zero.

$$
\begin{aligned}
&prod \equiv \\
&\qquad\quad (\lambda\ (m) \\
&\qquad\qquad (\lambda\ (n) \\
&\qquad\qquad\quad ((m\ (soma\ n))\ zero)))
\end{aligned}
$$

Note que *prod*, *soma*, *zero* e *succ* são **abreviações**.

Exponenciação

Como fazer?

$$
\begin{aligned}
&exp1 \equiv \\
&\qquad (\lambda\ (m) \\
&\qquad\quad (\lambda\ (n) \\
&\qquad\qquad ((n\ (prod\ m))\ um)))
\end{aligned}
$$

Segunda solução:

$$
\begin{aligned}
&exp2 \equiv \\
&\qquad (\lambda\ (m) \\
&\qquad\quad (\lambda\ (n) \\
&\qquad\qquad (n\ m)))
\end{aligned}
$$

Subtração

Predecessor: o truque é usar pares de valores, da seguinte forma:
(0 0), (0 1), (1 2), (2 3), $\cdots$

$$(\lambda\ (p)\quad (pair\quad (right\ p)\quad (succ\ (right\ p))))$$

Agora é possível definir um predecessor.

$$
\begin{array}{l}
(pred\ n) \equiv \\
\qquad (\lambda\ (n)\quad (left\ (\\
\qquad\qquad\qquad (n\quad (\lambda\ (p)\ (pair\ (right\ p)\ (succ\ (right\ p))))) \\
\qquad\qquad\qquad (pair\ zero\ zero) \\
\qquad\qquad\quad)) \\
\qquad)
\end{array}
$$

E $zero?$?

$$(\lambda\ (n)\quad ((n(\lambda\ (nada)\ no)\quad yes)))$$

Recursão

Dá para sumir com `define`?

Como fazer recursão sem usar recursão?

$$
\begin{aligned}
&fact \equiv\\
&\qquad (\lambda\ (n)\\
&\qquad\qquad (if\ \ (zero?\ n)\\
&\qquad\qquad\qquad 1\\
&\qquad\qquad\qquad (*\ n\ ({\color{red}\bullet}\ (sub1\, n)))))
\end{aligned}
$$

O ${\color{red}\bullet}$ significa a chamada recursiva. Repetir a definição não adianta, pela mesma razão que não funcionava com *environment* recursivo.

..continuação

Truque: usar um gerador.

mk-fact$\equiv (\lambda\ (f)$

$(\lambda\ (n)$

$(if\ (zero?\ n)$

1

$(*\ n\ (f\ (sub1\,n)))))$

O que acontece se fizermos f igual a mk-fact?

$(\lambda\ (n)$

$(if\ (zero?\ n)$

1

$(*\ n\ (\text{mk-fact}\ (sub1\,n)))))$

Cadê o argumento de mk-fact?

..continuação

Colocamos um argumento para f

$$
\begin{array}{l}
\text{mk-fact} \equiv (\lambda\ (f) \\
\qquad\qquad (\lambda\ (n) \\
\qquad\qquad\qquad (if\ (zero?\ n) \\
\qquad\qquad\qquad\quad 1 \\
\qquad\qquad\qquad\quad (*\ n\ ((f\ \textcolor{red}{\bullet})\ (sub1\, n)))))
\end{array}
$$

$$
\begin{array}{l}
(\text{mk-fact mk-fact}) \equiv \\
\qquad\qquad\qquad (\lambda\ (n) \\
\qquad\qquad\qquad\quad (if\ (zero?\ n) \\
\qquad\qquad\qquad\qquad 1 \\
\qquad\qquad\qquad\qquad (*\ n\ ((\text{mk-fact}\ \textcolor{red}{\bullet})\ (sub1\, n)))))
\end{array}
$$

Para $n \geq 2$, precisamos chamar $\textcolor{red}{\bullet}$

..continuação

E se $\bullet$ também for mk-fact?

$$
\begin{array}{l}
\text{mk-fact} \equiv (\lambda\ (f) \\
\qquad\qquad (\lambda\ (n) \\
\qquad\qquad\qquad (if\ (zero?\ n) \\
\qquad\qquad\qquad\quad 1 \\
\qquad\qquad\qquad\quad (*\ n\ ((f\ f)\ (sub1\, n))))))
\end{array}
$$

Funciona!

$$
\begin{array}{l}
\text{fact} \equiv ((\lambda\ (\text{mk-fact})\quad (\text{mk-fact mk-fact})) \\
\qquad (\lambda\ (f)\quad (\lambda\ (n) \\
\qquad\qquad\qquad (if\ (zero?\ n) \\
\qquad\qquad\qquad\quad 1 \\
\qquad\qquad\qquad\quad (*\ n\ ((f\ f)\ (sub1\, n)))))) \\
\qquad)
\end{array}
$$

..continuação

Escrito de uma forma diferente:

$$
\begin{array}{l}
(\\
\ (\lambda\ (\text{mk-fact})\quad (\text{mk-fact mk-fact})) \\
\ (\lambda\ (f) \\
\qquad\quad (\lambda\ (g) \\
\qquad\qquad\quad (\lambda\ (n) \\
\qquad\qquad\qquad\quad (if\ \ (zero?\ n) \\
\qquad\qquad\qquad\qquad\qquad\quad 1 \\
\qquad\qquad\qquad\qquad\qquad\quad (*\ n\ (g\ (sub1\ n)))))) \\
\qquad\quad (f\ \ f)) \\
))
\end{array}
$$

A parte em vermelho é quase a definição original.

Combinador Y

O operador que transforma uma função p em recursiva é:

$(\lambda\ (p)$

$($

$(\lambda\ (f)\ (f\ f))$

$(\lambda\ (f)\ (p\ (f\ f)))$

$))$

Mas tem um problema:recursão infinita...

$(\lambda\quad (p)$

$($

$(\lambda\ (f)\ (f\ f))$

$(\lambda\ (f)\ (p\ (\lambda\ (a)\ ((f\ f)\ a))))$

$))$