Write Yourself a Haskell... in Lisp
(por Gergö Érdi, traduzido por George Lukas)

Para mim, a melhor maneira de entender alguma coisa é, geralmente, para implementá-lo eu mesmo. Então, quando eu comecei a ficar sério sobre Haskell, eu implementei (em Common Lisp) uma linguagem de programação funcional preguiçosa simples que usa **graph rewriting** para a interpretação. Dez anos depois mencionei nessa publicação do Reddit (**mentioned this implementation**) sobre escrever um interpretador Lisp em Haskell (**writing a Lisp interpreter in Haskell**) (talvez inspirado pelo bem conhecido **Haskell tutorial** e os blogposts de mesmo nome). Então agora decidi limpar o velho código, focando em algo mais legível de se entender. O interpretador acabado tem menos de 500 linhas de Common Lisp (não incluindo o type checker que será o tema do meu próximo post).
Gostaria também de mencionar que é lógico que não vamos realmente implementar Haskell aqui. Como veremos, a linguagem é um modelo muito simplificado de linguagens funcionais puras, preguiçosas. É realmente mais algo mais próximo do núcleo do GHC do que do Haskell propriamente dito.

Syntax
Nós não preocuparemos com o parsing: nossos programas de entrada serão S-expressions. Por exemplo:

```
data List a = Nil | Cons a (List a)

map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Nós escreveremos:

```
(defdata list (a) nil (cons a (list a)))

(deffun map
  ((f nil) nil)
  ((f (cons x xs)) (cons (f x) (map f xs))
```

Como podemos ver não há distinção sintática entre nomes de variáveis e nomes de construtores — quanto traduzimos essas S-expressions, nós tomaremos o cuidado especial de sempre registrar o construtor de nomes antes de processar as expressões.
Internamente, nós desempenharemos programas em um método mais simplificado, fazendo todas as aplicações de funções de forma explicita. Multi-parameter, funções e constructors serão implementados via **schönfinkeling**. A árvore da sintaxe é representada usando duas hierarquias de classes: subclasses de *expr* para expressões e subclasses de *pattern* para patterns:

```lisp
(defclass expr ()
  ()
  (:documentation "Expression"))

(defclass symbol-expr (expr)
  ((symbol :initarg :symbol :reader expr-symbol)))

(defclass var-expr (symbol-expr)
  ()
  (:documentation "Variable occurance"))

(defclass cons-expr (symbol-expr)
  ()
  (:documentation "Constructor"))

(defclass apply-expr (expr)
  ((fun :initarg :fun :reader expr-fun)
   (arg :initarg :arg :reader expr-arg))
  (:documentation "Function application"))

(defclass let-expr (expr)
  ((bindings :initarg :bindings :reader expr-bindings)
   (body :initarg :body :reader expr-body))
  (:documentation "Let"))

(defclass lambda-expr (expr)
  ((formals :initarg :formals :reader expr-formals)
   (body :initarg :body :reader expr-body))
  (:documentation "Lambda abstraction"))

(defclass pattern ()
  ()
  (:documentation "Pattern"))

(defclass wildcard-pattern (pattern)
  ()
  (:documentation "Wildcard pattern"))

(defclass var-pattern (pattern)
  ((symbol :initarg :symbol :reader pattern-symbol))
  (:documentation "Variable binder"))

(defclass cons-pattern (pattern)
  ((cons :initarg :cons :reader pattern-cons)
   (args :initarg :args :initform nil :reader pattern-args))
  (:documentation "Constructor pattern"))
```

*syntax-defs.lisp

Indo a partir de S-expressions para esta definição orientada a objetos é simples:

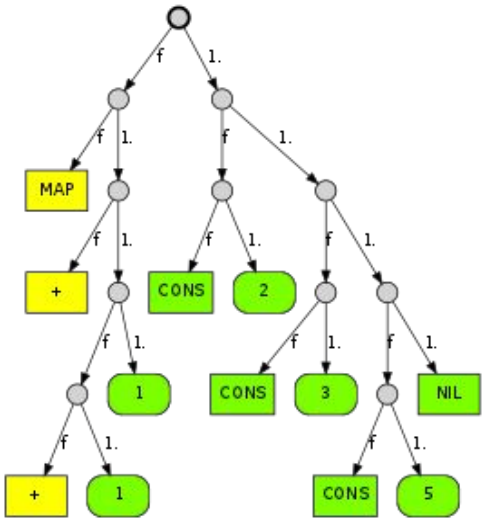
```
(defun normalize-pat (pat)
  (typecase pat
    (atom
     (cond ((eq pat :wildcard) (make-instance 'wildcard-pattern))
           ((constructorp pat) (make-instance 'cons-pattern :cons pat))
           (t                  (make-instance 'var-pattern :symbol pat))))
    (cons
     (make-instance 'cons-pattern
                    :cons (first pat)
                    :args (mapcar #'normalize-pat (rest pat))))))

(defun normalize-expr (expr)
  (typecase expr
    (atom
     (make-instance (if (constructorp expr) 'cons-expr 'var-expr) :symbol expr))
    (cons
     (case (first expr)
       ((let)
        (destructuring-bind ((&rest bindings) body) (rest expr)
          (make-instance 'let-expr
                         :bindings (loop for (name val) in bindings
                                         collect (cons name (normalize-expr val)))
                         :body (normalize-expr body))))
       ((lambda)
        (destructuring-bind ((&rest patterns) body) (rest expr)
          (make-instance 'lambda-expr
                         :formals (mapcar #'normalize-pat patterns)
                         :body (normalize-expr body))))
       (otherwise
        (reduce (lambda (f x) (make-instance 'apply-expr :fun f :arg x))
                (mapcar #'normalize-expr expr)))))))
```

*syntax.lisp

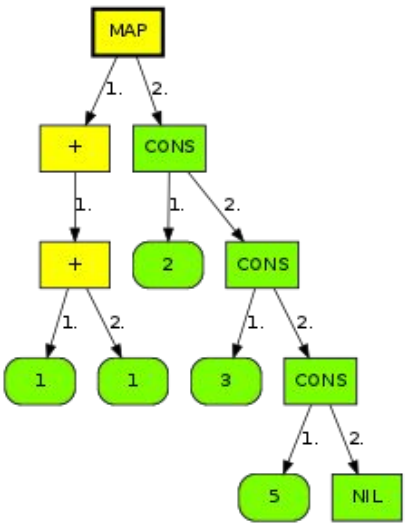
Graph rewriting semantics

A idea básica por tráz de implementar uma linguagem funcional preguiçosa é usar graph rewriting para representar termos diretamente como grafos e a redução da aplicação da função e a redução de um aplicativo função é implementada através da substituição do nó da aplicação com o grafo correspondente para o lado direito da definição da função, é claro, com todas as referências a argumentos formais instanciado para os argumentos reais. Vejamos primeiramente um exemplo:

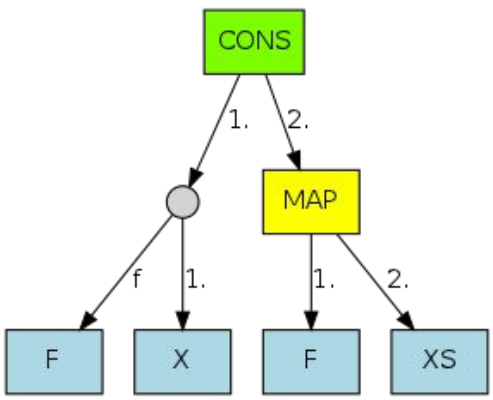
```
(defvar x
  (let ((primes (cons 2 (cons 3 (cons 5 nil)))))
    (map (+ (+ 1 1) primes))))
```



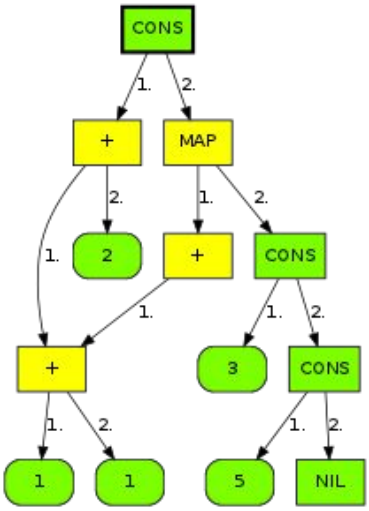
Como podemos ver, as caixas verdes representan construtores (As you can see, the green boxes represent constructors (com cantos arredondados para valores primitivos), as amareladas são funções e os círculos acinzentados são aplicações de funções. Não há variáveis, uma vez que as referencias são resolvidas usando esse grafo. Nós podems simplificar esse formato omitindo os nós de aplicação e apenas empurrar os argumentos perante seu nó (até que se torne saturado), como a seguir:



Reduzi-lo implica instanciar o grafo a seguir (a partir do segundo caso do map), com f ligado a (+ (+ 1 1), x ligado a 2, e xs ligado a (cons 3 (cons 5 nil)). Note o nó de explícito: nós ainda não sabemos a aridade de f até que seja ligado a alguma coisa.

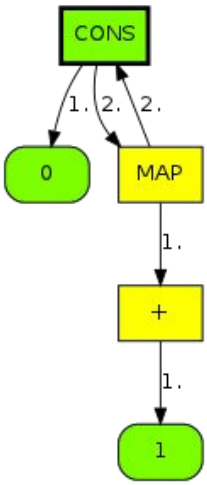


Resulta em:

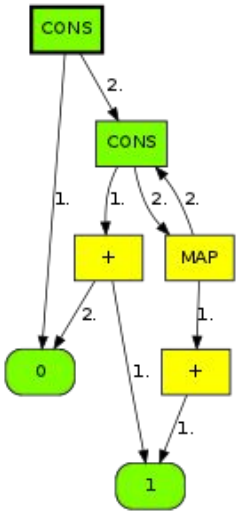


Note como o primeiro argumento para + é compartilhado no resultado. Agora note como (+ (+ 1 1) 2) não é reduzido pelo map, uma vez que inerte em seu primeiro argumento. Nosso segundo exemplo mostra termos recursivamente definidos:

```
(defvar nats (cons 0 (map (+ 1) nats)))
```



Isto já está em Weak Head Normal Form (**WHNF**), mas no entanto, reduzir a cauda da lista, para obter esse resultado:



Decorrentemente.

Representando os grafos

Uma vez que queremos substituir facilmente subgrafos de funções de aplicações application (quando reduzimos essas aplicações), a representação usa um nivel extra de indireção: gnodes contém a carga: o tipo do nó e os pontos para seus filhos. grefs; e cada gref contém uma única referencia para seu nó de conteúdo. grefs podem ser compartilhados, então por isso, quando o seu conteúdo gnode for substituído, todas as referências compartilhadas são atualizadas.

```
(defclass dotable ()
  ((dot-name :initform (gensym "V") :reader dot-name))
  (:documentation "Something that can be turned into a GraphViz node"))

(defclass gnode (dotable)
  ()
  (:documentation "Node in the graph representation"))

(defclass gref (dotable)
  ((node :initarg :node :accessor gderef))
  (:documentation "Pointer in  the graph representation"))

(defun make-gref (gnode)
  (make-instance 'gref :node gnode))

(defclass cons-gnode (gnode)
  ((cons :initarg :cons :reader gnode-cons)
   (args :initarg :args :initform nil :accessor gnode-args)))

(defclass param-gnode (gnode)
  ((var :initarg :var :reader gnode-var)))
```

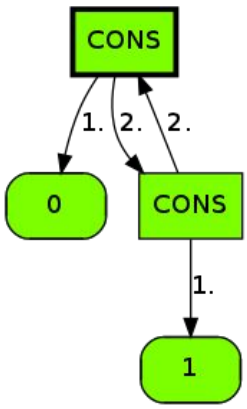
```
(defclass apply-gnode (gnode)
  ((fun :initarg :fun :accessor gnode-fun)
   (args :initarg :args :accessor gnode-args)))

(defclass fun-gnode (gnode)
  ((fun-name :initarg :fun-name :reader gnode-fun-name)
   (arity :initarg :arity :reader gnode-fun-arity)
   (args :initarg :args :initform (list) :accessor gnode-args)))
```

*graph-defs.lisp

Variable nodes

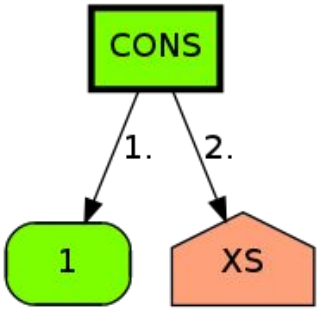
Anteriormente determinamos que variáveis não eram presentes, tal como sua representação em grafos uma vez que elas são , uma vez que eles são sequenciais (respeitando o compartilhamento é claro). Mas ainda temos as classes avar-gnode class definidas acima. A razão para isso é simplesmente as ocorrencias de variáveis formais em definições de funções, que serão preenchidos quando reduzimos as aplicações de função.
Mas ainda há um problemas mais complicado com variáveis, nossa linguagem let-constructor é mutuamente recursiva, então por isso não podemos criar grafos das variáveis uma por uma:

```
(defvar main
  (let ((zig (cons 0 zag))
        (zag (cons 1 zig)))
    zig))
```



É claro, nem tudo pode ser ajudado por esse:

```
(defvar silly-list
  (let ((xs xs))
    (cons 1 xs)))
```



Então nós adicionamos uma subclasse para armazenar temporáriamente as ocorrências da variável let-bound e substituímos-a após o fato. Infelizmente isso também ocorre por causa que o codigo tradúz a partir de *expr* para *gnode* para se tornar um pequeno spaghetti-y. Por outro lado nós ainda não temos implementado o *lambda lifting*, como não é estritamente essêncial — o programador terá que implementar isso por si só, por escrever top-level functions.

```
(defgeneric graph-from-expr (expr))

(defmethod graph-from-expr ((expr var-expr))
  (let* ((var-name (expr-symbol expr))
         (var-data (cdr (assoc var-name *vars*))))
    (if var-data (graph-from-var var-name var-data)
        (make-gref
         (or (let ((fun (lookup-function var-name)))
               (and fun (make-instance 'fun-gnode :fun-name var-name :arity (function-arity fun))))
             (make-instance 'param-gnode :var var-name))))))

(defmethod graph-from-expr ((expr cons-expr))
  (make-gref (make-instance 'cons-gnode :cons (expr-symbol expr))))

(defmethod graph-from-expr ((expr apply-expr))
  (make-gref
   (let ((fun (graph-from-expr (expr-fun expr)))
         (arg (graph-from-expr (expr-arg expr) )))
     (make-instance 'apply-gnode :fun fun :args (list arg)))))

(defmethod graph-from-expr ((expr let-expr))
  (let ((*vars* *vars*))
    (add-vars (expr-bindings expr))
    (fill-var-gref (graph-from-expr (expr-body expr)))))

(defmethod graph-from-expr ((expr lambda-expr))
  (error "Not implemented: lambdas"))

(defvar *vars-built*)

(defclass bottom-gnode (gnode)
  ((var :initarg :var :reader gnode-var)
   (ptr :initarg :ptr :reader gnode-var-ptr)))
```

```
(defun graph-from-var (var-name var-data)
  (or (car (second var-data))
      (let* ((gref (make-instance 'gref))
             (gref-ptr (list gref)))
        (setf (gderef gref) (make-instance 'bottom-gnode :var var-name :ptr gref-ptr))
        (setf (second var-data) gref-ptr)
        (let ((gref* (graph-from-expr (first var-data))))
          (rplaca gref-ptr gref*)
          gref*))))

(defvar *filled*)

(defun fill-var-gref (gref)
  (let ((*filled* (list)))
    (fill-var-gref* gref)))

(defgeneric fill-var-gref** (gref gnode))

(defmethod fill-var-gref** (gref (gnode bottom-gnode))
  (let ((gref* (car (gnode-var-ptr gnode))))
    (fill-var-gref* gref*)))

(defmethod fill-var-gref** (gref gnode)
  (fill-var-gnode (gderef gref))
  gref)

(defun fill-var-gref* (gref)
  (or (cdr (assoc gref *filled*))
      (progn
        (push (cons gref gref) *filled*)
        (let ((gref* (fill-var-gref** gref (gderef gref))))
          (push (cons gref gref*) *filled*)
          gref*))))

(defgeneric fill-var-gnode (gnode))

(defmethod fill-var-gnode ((gnode gnode))
  (values))

(defmethod fill-var-gnode ((gnode cons-gnode))
  (setf (gnode-args gnode) (mapcar #'fill-var-gref* (gnode-args gnode)))
  (values))

(defmethod fill-var-gnode ((gnode apply-gnode))
  (setf (gnode-fun gnode) (fill-var-gref* (gnode-fun gnode)))
  (setf (gnode-args gnode) (mapcar #'fill-var-gref* (gnode-args gnode)))
  (values))
```

*graph.lisp

Making it tick

Há três partes para fazer reduções trabalho; em primeiro lugar, uma maneira de fazer a correspondência de padrões em relação às alternativas de função. Em segundo lugar, dado um mapeamento para este match, instancia-se uma função por substuição dos var-gnodes. A terceira parte está orquestrado para qual parte para cuidar da parte de escolher quais nós reduzimos.
Pattern matching é relativamente simples: uma um padrão variável sempre tem sucesso (e vincula o subgrafo) e um padrão de construtor pode ter sucesso e recursivamente, falha, ou (se o nó atual é uma aplicação de função) força a redução. Isso será feito posteriormente através de uma condição Lisp (**condition**).

```
(define-condition no-match ()
  ())

(define-condition need-reduce ()
  ((gref :initarg :gref :reader need-reduce-gref))
  (:report (lambda (condition stream)
          (format stream "Need to reduce ~A" (need-reduce-gref condition)))))

(defgeneric match-pattern* (pat gnode gref))

(defun match-pattern (pat gref)
  (match-pattern* pat (gderef gref) gref))

(defmethod match-pattern* ((pat wildcard-pattern) gnode gref)
  (list))

(defmethod match-pattern* ((pat var-pattern) gnode gref)
  (list (cons (pattern-symbol pat) gref)))

(defmethod match-pattern* ((pat cons-pattern) (gnode cons-gnode) gref)
  (unless (eq (pattern-cons pat) (gnode-cons gnode))
    (error 'no-match))
  ;; Because of well-typedness, there is no need to check arity here
  (mapcan #'match-pattern (pattern-args pat) (gnode-args gnode)))

(defmethod match-pattern* ((pat cons-pattern) gnode gref)
  (error 'need-reduce :gref gref))
```

*match.lisp

Uma vez que temos as ligações no formato retornado por match-pattern, podemos facilmente instanciar corpos de função, nós apenas temos que manter um mapeamento a partir do nó antigo para o novo, para evitar divergências em ciclos.

```
(defgeneric deepclone-gnode (gnode vars mapping))

(defmethod deepclone-gnode ((gnode cons-gnode) vars mapping)
  (make-instance 'cons-gnode
                 :cons (gnode-cons gnode)
                 :args (loop for arg in (gnode-args gnode)
                             collect (deepclone-gref arg vars mapping))))

(defmethod deepclone-gnode ((gnode param-gnode) vars mapping)
  gnode)

(defmethod deepclone-gnode ((gnode apply-gnode) vars mapping)
  (make-instance 'apply-gnode
                 :fun (deepclone-gref (gnode-fun gnode) vars mapping)
                 :args (loop for arg in (gnode-args gnode)
                             collect (deepclone-gref arg vars mapping))))

(defmethod deepclone-gnode ((gnode fun-gnode) vars mapping)
  (make-instance 'fun-gnode
                 :fun-name (gnode-fun-name gnode)
                 :arity (gnode-fun-arity gnode)
                 :args (loop for arg in (gnode-args gnode)
                             collect (deepclone-gref arg vars mapping))))

(defun deepclone-gref (gref &optional vars (mapping (make-hash-table)))
  (or
   (and (typep (gderef gref) 'param-gnode)
        (cdr (assoc (gnode-var (gderef gref)) vars)))
   (gethash gref mapping)
   (let ((gref/copy (make-instance 'gref)))
     (setf (gethash gref mapping) gref/copy)
     (setf (gderef gref/copy) (deepclone-gnode (gderef gref) vars mapping))
     gref/copy)))
```

*deepclone.lisp

A redução em seguida, torna-se apenas uma questão de colocar esses dois módulos juntos. Várias funções são fornecidas com diferentes granularidades: reduce-graph tenta reduzir diretamente, reduce-graph* obtem recursivamente as condições de need-reduce nesses nós, com efeito certificando-se de um passo único de redução no local alvo pode ser feita; e reduce-to-whnf repetidamente usa reduce-graph* até que a cabeça seja um construtor ou uma aplicação de função não saturado. simplify-apps não é necessáriamente um passo de redução, isso apenas remove supérfluos em apply-gnodes.

```
(defgeneric gnode-force-cons (gnode gref))
```

```
(defmethod gnode-force-cons ((gnode cons-gnode) gref)
  (gnode-cons gnode))

(defmethod gnode-force-cons ((gnode gnode) gref)
  (error 'need-reduce :gref gref))

(defgeneric reduce-function (fun-info args))

(defmethod reduce-function ((fun-info prim-function-info) arg-grefs)
  (let ((args (loop for arg-gref in arg-grefs
                    collect (gnode-force-cons (gderef arg-gref) arg-gref))))
    (make-instance 'cons-gnode
                   :cons (apply (prim-function fun-info) args))))

(defmethod reduce-function ((fun-info match-function-info) args)
  (loop for (pats . body) in (function-matches fun-info)
        do (handler-case (return (gderef (deepclone-gref body (mapcan #'match-pattern pats args))))
             (no-match ()))
        finally (error 'no-match)))

(defgeneric reduce-graph-node (gnode))

(defun reduce-graph (gref)
  (let ((new-gnode (reduce-graph-node (gderef gref))))
    (when new-gnode
      (setf (gderef gref) new-gnode)
      t)))

(defun reduce-graph* (gref)
  (prog1
      (handler-case (reduce-graph gref)
        (need-reduce (req)
          (when (reduce-graph* (need-reduce-gref req))
            (reduce-graph* gref))))
    (simplify-apps gref)))

(defun reduce-to-whnf (gref)
  (when (reduce-graph* gref)
    (reduce-to-whnf gref)))

(defmethod reduce-graph-node ((gnode cons-gnode))
  nil)

(defmethod reduce-graph-node ((gnode bottom-gnode))
  nil)

(defmethod reduce-graph-node ((gnode param-gnode))
  (error "Internal error: parameter reference in head"))

(defun split-at (lst i)
  (labels ((aux (lst i before)
```

```
          (cond
            ((null lst) (values (reverse before) nil (= i 0)))
            ((= i 0)    (values (reverse before) lst t))
            (t          (aux (cdr lst) (1- i) (cons (car lst) before))))))
    (aux lst i nil)))

(defmethod reduce-graph-node ((fun fun-gnode))
  (multiple-value-bind (actuals remaining saturated) (split-at (gnode-args fun) (gnode-fun-arity fun))
    (when saturated
      (let* ((fun-info (lookup-function (gnode-fun-name fun)))
             (result-gnode (reduce-function fun-info actuals)))
        (if (null remaining) result-gnode
            (make-instance 'apply-gnode :fun (make-gref result-gnode) :args remaining))))))

(defgeneric gnode-add-args (gnode args))

(defmethod gnode-add-args ((gnode gnode) args)
  nil)

(defmethod gnode-add-args ((gnode cons-gnode) args)
  (make-instance 'cons-gnode
                 :cons (gnode-cons gnode)
                 :args (append (gnode-args gnode) args)))

(defmethod gnode-add-args ((gnode fun-gnode) args)
  (make-instance 'fun-gnode
                 :fun-name (gnode-fun-name gnode)
                 :arity (gnode-fun-arity gnode)
                 :args (append (gnode-args gnode) args)))

(defmethod gnode-add-args ((gnode apply-gnode) args)
  (make-instance 'apply-gnode
                 :fun (gnode-fun gnode)
                 :args (append (gnode-args gnode) args)))

(defmethod reduce-graph-node ((app apply-gnode))
  (let ((fun (gnode-fun app)))
    (or (gnode-add-args (gderef fun) (gnode-args app))
        (error 'need-reduce :gref fun))))

(defgeneric simplify-app (gnode))

(defmethod simplify-app ((gnode gnode))
  nil)

(defmethod simplify-app ((app apply-gnode))
  (mapcar #'simplify-app-gref (cons (gnode-fun app) (gnode-args app)))
  (gnode-add-args (gderef (gnode-fun app)) (gnode-args app)))

(defvar *grefs*)
```

```
(defun simplify-app-gref (gref)
  (unless (gethash gref *grefs*)
    (setf (gethash gref *grefs*) t)
    (let ((new-gnode (simplify-app (gderef gref))))
      (when new-gnode
        (setf (gderef gref) new-gnode)))))

(defun simplify-apps (gref)
  (let ((*grefs* (make-hash-table)))
    (simplify-app-gref gref)))
```

*reduce.lisp

Housekeeping

O resto do código apenas mantém um registro de funções e construtores e define algumas funções primitivas (note como nós precisamos de um simples bool ADT para ser um built-in, apenas por isso temos um tipo de retorno para >=).

```
(defvar *constructors*)
(defvar *functions*)
(defvar *vars*)

(defmacro in-fresh-context (&body body)
  `(let ((*constructors* (make-hash-table))
         (*functions* (make-hash-table))
         (*vars* (list)))
     (register-builtin-types)
     (register-prim-functions)
     ,@body))

(defun add-constructors (constructors)
  (loop for (cons-name . cons-type) in constructors
        do (setf (gethash cons-name *constructors*) cons-type)))

(defun add-vars (vars)
  (setf *vars*
        (append
         (loop for (var-name . var-expr) in vars
               collect (list var-name var-expr nil))
         *vars*)))




(defun constructorp (x)
  (typecase x
    (integer t)
```

```
    (string t)
    (atom
     (nth-value 1 (gethash x *constructors*)))))

(defclass function-info ()
  ())

(defclass proto-function-info (function-info)
  ((arity :initarg :arity :reader function-arity)))

(defclass prim-function-info (proto-function-info)
  ((fun :initarg :fun :reader prim-function)))

(defclass match-function-info (function-info)
  ((matches :initarg :matches :reader function-matches)))

(defmethod function-arity ((function-info match-function-info))
  (length (caar (function-matches function-info))))

(defun register-proto-function (fun-name arity)
  (setf (gethash fun-name *functions*)
        (make-instance 'proto-function-info :arity arity)))

(defun register-match-function (fun-name matches)
  (setf (gethash fun-name *functions*)
        (make-instance 'match-function-info :matches matches)))

(defun register-prim-function (fun-name arity fun)
  (setf (gethash fun-name *functions*)
        (make-instance 'prim-function-info :arity arity :fun fun)))

(defun lookup-function (fun-name)
  (format nil "~A" *functions*)
  (gethash fun-name *functions*))
```

*registry.lisp

```
(defun register-builtin-types ()
  (add-constructors '((true . bool)
                      (false . bool))))

(defun register-prim-functions ()
```

```
(labels
   ((pred (p)
      (lambda (&rest args)
        (if (apply p args) 'true 'false))))
 (register-prim-function '>= 2 (pred #'>=))
 (register-prim-function '+ 2 #'+)
 (register-prim-function '- 2 #'-)
 (register-prim-function '* 2 #'*)
 (register-prim-function '/ 2 #'/)
 (register-prim-function 'show-int 1 (lambda (x) (format nil "~D" x)))
 (register-prim-function 'string-append 2 (lambda (x y) (format nil "~A~A" x y)))))
```

*prim.lisp

Juntando tudo

Dada uma S-expression contendo um programa de Alef, podemos transformá-lo em um gráfico adequado para redução, fazendo o seguinte etapas:

1. Quebrar em definições de tipo de dados, definições de variáveis e definições de função
2. Processar definições de datatypes por registo dos nomes de construtores (uma vez que ainda não fazemos o typechecking, a única informação necessária sobre os construtores é que eles não são chamadas de função/referências variáveis)
3. Registrar nomes de variáveis de alto nivel
4. Registrar nomes de funções
5. Processar definições de variáveis de alto nível em grafos
6. Processar definições de funções em templates de grafos
7. Devolver o gráfico da variável de alto nível para a chamada main como o programa atual.

```
(defun partition-by (f xs)
 (let ((table (make-hash-table)))
  (loop for x in xs
      for key = (funcall f x)
      do (push x (gethash key table)))
  (maphash (lambda (key group) (setf (gethash key table) (reverse group))) table)
  table))
```

```
(defun parse-program (program)
 (let ((program-parts (partition-by #'first program)))

  (add-constructors
   (loop for (defdata type-name type-args . constructors) in (gethash 'defdata program-parts)
      append (loop for (cons-name . cons-args) in constructors
```

```
                collect (cons cons-name type-name))))

  (add-vars
   (loop for (defvar var-name var-body) in (gethash 'defvar program-parts)
       collect (cons var-name (normalize-expr var-body))))

  (let ((fun-defs
       (loop for (deffun name . matches) in (gethash 'deffun program-parts)
           do (register-proto-function name (length (caar matches)))
           collect (cons name (loop for (pats expr) in matches
                                    collect (cons (mapcar #'normalize-pat pats) (normalize-expr expr)))))))

    ;; Build graphs for functions
    (loop for (fun-name . matches) in fun-defs
        for match-nodes = (loop for (pats . expr) in matches
                                collect (cons pats (fill-var-gref (graph-from-expr expr))))
        do (register-match-function fun-name match-nodes)))

  (fill-var-gref (graph-from-var 'main (cdr (assoc 'main *vars*))))))
```

*program.lisp

Visualização

As agradáveis plotagens deste blogpost foram criados pelo dumping gráfico do programa em formato GraphViz, e rodando na saída **dot**. O código de visualização é um percurso simples do gráfico. Note como nós guardamos um nome estável (gerado usando Gensym) em cada nó, para garantir uma correspondência nos nós Graphviz gerados entre as etapas de redução. No futuro, poderiamos usado-lo para, por exemplo animar o gráfico para mostrar cada etapa de redução.

```
(defvar *dot-mapping*)
(defvar *dot-stream*)

(defparameter *visualize-grefs* nil)

(defgeneric to-dot (x rootp))
```

```
(defmethod to-dot ((gref gref) rootp)
  (or (gethash gref *dot-mapping*)
      (let ((dot (dot-name (if *visualize-grefs* gref (gderef gref)))))
        (setf (gethash gref *dot-mapping*) dot)
        (to-dot (gderef gref) rootp)
        (when *visualize-grefs*
          (dot-node (dot-name gref) "" "shape=point")
```

```
        (dot-edge (dot-name gref) (dot-name (gderef gref)) ""))
      dot)))

(defun format-symbol (x)
  (if (symbolp x) (symbol-name x)
    (format nil "~S" x)))

(defconstant +dot-escape-chars+ "\"")

(defun dot-escape (str)
  (labels ((escape-p (char)
             (find char +dot-escape-chars+))

           (escape-char (char)
             (format nil "\\~A" char)))
    (with-output-to-string (s)
      (loop for start = 0 then (1+ pos)
            for pos = (position-if #'escape-p str :start start)
            do (write-sequence str s :start start :end pos)
            when pos do (write-sequence (escape-char (char str pos)) s)
            while pos))))

(defun dot-node (dot label rootp &optional style)
  (format *dot-stream* "~&~A [label=\"~A\" ~A ~A]"
          dot
          (dot-escape label)
          (if rootp "penwidth=3" "")
          (if style (format nil ", ~A" style) "")))

(defun dot-edge (dot/from dot/to label)
  (format *dot-stream* "~&~A -> ~A[label=\"~A\"]" dot/from dot/to (dot-escape label)))

(defmethod to-dot ((gnode bottom-gnode) rootp)
    (dot-node (dot-name gnode) (format-symbol (gnode-var gnode)) rootp "shape=house, fillcolor=lightsalmon")
  ;; (dot-node dot "" "shape=house, fillcolor=lightsalmon")
  ;; (dot-node dot "" "shape=point")
  ;; (dot-edge dot dot "")
  )




(defmethod to-dot ((gnode cons-gnode) rootp)
  (let* ((prim (typecase (gnode-cons gnode)
                 (integer t)
                 (string t)))
         (style (if prim "\"filled, rounded\"" "filled"))
         (color "chartreuse")
         (style (format nil "shape=box, style=~A, fillcolor=~A" style color)))
```

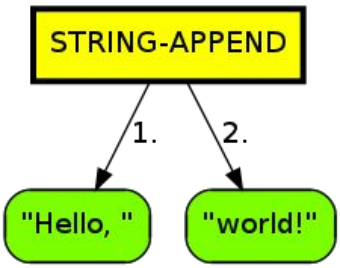
```
   (dot-node (dot-name gnode) (format-symbol (gnode-cons gnode)) rootp style))
  (loop for arg-dot in (mapcar #'(lambda (x) (to-dot x nil)) (gnode-args gnode))
        for index = 1 then (1+ index)
        do (dot-edge (dot-name gnode) arg-dot (format nil "~D." index))))

(defmethod to-dot ((gnode param-gnode) rootp)
      (dot-node (dot-name gnode) (format-symbol (gnode-var gnode)) rootp "shape=box, fillcolor=lightblue"))

(defmethod to-dot ((gnode apply-gnode) rootp)
  (dot-node (dot-name gnode) "" rootp "shape=circle, ordering=out, fixedsize=true, width=.25")
  (dot-edge (dot-name gnode) (to-dot (gnode-fun gnode) nil) "f")
  (loop for arg-dot in (mapcar #'(lambda (x) (to-dot x nil)) (gnode-args gnode))
        for index = 1 then (1+ index)
        do (dot-edge (dot-name gnode) arg-dot (format nil "~D." index))))

(defmethod to-dot ((gnode fun-gnode) rootp)
    (dot-node (dot-name gnode) (format-symbol (gnode-fun-name gnode)) rootp "shape=box, fillcolor=yellow")
  (loop for arg-dot in (mapcar #'(lambda (x) (to-dot x nil)) (gnode-args gnode))
        for index = 1 then (1+ index)
        do (dot-edge (dot-name gnode) arg-dot (format nil "~D." index))))

(defun dot-from-graph (gref &optional (stream *standard-output*))
  (let ((*dot-mapping* (make-hash-table))
        (*dot-stream* stream))
    (format *dot-stream* "digraph G{")
    (format *dot-stream* "~&graph[size=\"3,3\", bgcolor=\"transparent\"]")
    (format *dot-stream* "~&node[style=filled]")
    (to-dot gref t)
    (format *dot-stream* "~&}")))
```

*visualize.lisp

Podemos usar essa forma:

```
(in-fresh-context
  (let ((g (parse-program '((defvar main (string-append "Hello, " "world!"))))))
    (simplify-apps g)
    (with-open-file (s "hello.dot" :direction :output :if-exists :supersede)
      (dot-from-graph g s))
    (reduce-to-whnf g)
    (with-open-file (s "hello-reduced.dot" :direction :output :if-exists :supersede)
      (dot-from-graph g s))))
```

O que resulta em gráficos descartáveis::



E:

Conclusão

SLOCCount me disse o código apresentado neste blogpost contém 471 linhas em Common Lisp; você pode verificar o código-fonte completo no **GitHub**. Ele implementa semântica preguiçosa em uma linguagem de programação funcional pura com correspondência de padrão em tipos de dados algébricos; por fim, alterando a definição de reduce-function, poderíamos facilmente torná-la estricta.