Curry

A Tasty dish?

Haskell Curry!
Curried Functions

• Currying is a functional programming technique that takes a function of N arguments and produces a related one where some of the arguments are fixed

• In Scheme
  – (define add1 (curry + 1))
  – (define double (curry * 2))
A tasty dish?

- Currying was named after the Mathematical logician Haskell Curry (1900-1982)
- Curry worked on combinatory logic ...
- A technique that eliminates the need for variables in mathematical logic ...
- and hence computer programming!
  - At least in theory
- The functional programming language Haskell is also named in honor of Haskell Curry
Functions in Haskell

• In Haskell we can define \( g \) as a function that takes two arguments of types \( a \) and \( b \) and returns a value of type \( c \) like this:
  
  \[- g :: (a, b) \rightarrow c \]

• We can let \( f \) be the curried form of \( g \) by
  
  \[- f = \text{curry} \ g \]

• The function \( f \) now has the signature
  
  \[- f :: a \rightarrow b \rightarrow c \]

• \( F \) takes an arg of type \( a \) & returns a function that takes an arg of type \( b \) & returns a value of type \( c \)
Functions in Haskell

• All functions in Haskell are curried, i.e., all Haskell functions take just single arguments.

• This is mostly hidden in notation, and is not apparent to a new Haskeller

• Let's take the function `div :: Int -> Int -> Int` which performs integer division

• The expression `div 11 2` evaluates to 5

• But it's a two-part process
  – `div 11` is evaled & returns a function of type `Int -> Int`
  – That function is applied to the value 2, yielding 5
Currying in Scheme

• Scheme has an explicit built in function, curry, that takes a function and some of its arguments and returns a curried function.

• For example:
  – (define add1 (curry + 1))
  – (define double (curry * 2))

• We could define this easily as:
  (define (curry fun . args)
    (lambda x (apply fun (append args x)))))
Note on lambda syntax

• (lambda X (foo X)) is a way to define a lambda expression that takes any number of arguments

• In this case X is bound to the list of the argument values, e.g.:
  > (define f (lambda x (print x)))
  > f
  #<procedure:f>
  > (f 1 2 3 4 5)
  (1 2 3 4 5)>
A real world example

• This weekend I wanted to adapt an example written for Lisp by Peter Norvig
• It is a very simple program that generates random sentences given a context free grammar
• It was written to take the grammar and start symbol as global variables 😞
• I wanted to make this a parameter, but it made the code more complex 😞 😞
• Scheme’s curry helped solve this!
Simple example

• Compare two lists of numbers pair wise:
  (apply and (map < ‘(0 1 2 3) ‘(5 6 7 8)))

• Is every number in a list positive?
  (apply and (map < 0 ‘(5 6 7 8)))

• Use (lambda (x) (< 0 x)) as the function
  (apply and (map (lambda (x) (< 0 x)) ‘(5 6 7 8)))

  ▪ Or use (curry < 0)
    (apply and (map (curry < 0) ‘(5 6 7 8)))
#lang scheme  

;;; This is a simple ...

(provide generate)

(define grammar
  '((S -> (NP VP) (NP VP) (NP VP) (NP VP) (NP VP) (NP VP) (S CONJ S)))
   (NP -> (ARTICLE ADJS? NOUN PP?))
   (VP -> (VERB NP) (VERB NP) (VERB NP) VERB)
   (ARTICLE -> the the the a a a one every)
   (NOUN -> man ball woman table penguin student book dog worm computer robot )
   ...
))
(define (generate phrase)
  ;; generate a random sentence or phrase from grammar
  (cond ((list? phrase)
         (apply append (map generate phrase)))
        ((non-terminal? phrase)
         (generate (random-element (rewrites phrase))))
        (#t (list phrase)))

(define (non-terminal? x)
  ;; True iff x is a on-terminal in grammar
  (assoc x grammar))

(define (rewrites non-terminal)
  ;; Return a list of the possible rewrites for non-terminal in grammar
  (rest (rest (assoc non-terminal grammar))))

(define (random-element list)
  ;; returns a random top-level element from list
  (list-ref list (random (length list)))))
Welcome to MzScheme v4.2.4 ...

> (require "cfg1.ss")

> (generate 'S)

(a woman took every mysterious ball)

> (generate 'S)

(a blue man liked the worm over a mysterious woman)

> (generate 'S)

(the large computer liked the dog in every mysterious student in the mysterious dog)

> (Generate 'S)

(the hungry table hit a worm under every mysterious blue penguin)

> (generate 'S)

(the book with a large large dog liked a ball)
((define default-grammar
  '((S -> (NP VP) (NP VP) (NP VP) (NP VP) ...))))

(define default-start 'S)

(define (generate (grammar default-grammar) (phrase default-start))
  ;; generate a random sentence or phrase from grammar
  (cond ((list? phrase)
         (apply append (map (curry generate grammar) phrase)))
        ((non-terminal? phrase grammar)
         (generate grammar (random-element (rewrites phrase grammar))))
        (#t (list phrase))))

(define (non-terminal? x grammar)
  ;; True iff x is a on-terminal in grammar
  (assoc x grammar))

(define (rewrites non-terminal grammar)
  ;; Return a list of the possible rewrites for non-terminal in grammar
  (rest (rest (assoc non-terminal grammar))))
> (require "cfg2.ss")
> (generate)
(\text{one ball across one man hated})
> (define g2 '[(S -> (a S b) (a S b) (a S b) ())])
> (generate g2)
(a a a a a a b b b b b b)
> (generate g2)
(a a a a a a a a a a a a b b b b b b b b b b b b)
> (generate g2)
()