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 <u>Haskell Curry</u> (1900-1982)
- Curry worked on combinatory logic ...
- A technique that eliminates the need for variables in <u>mathematical logic</u> ...
- and hence computer programming!
 - At least in theory
- The functional programming language <u>Haskell</u> 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) -> c
- We can let f be the curried form of g by
 - $-f = \underline{\text{curry}} g$
- The function f now has the signature
 - -f::a->b->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 <u>div</u> :: <u>Int</u> -> <u>Int</u> -> <u>Int</u> 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
 - #rocedure:f>
 - > (f 1 2 3 4 5)
 - (12345)
 - _

Simple example (a)

- Compare two lists of numbers pair wise:
 (apply and (map < '(0 1 2 3) '(5 6 7 8)))
- Note that (map < '(0 1 2 3) '(5 6 7 8)) evaluates to the list (#t #t #t #t)
- · Applying and to this produces the answer, #t

Simple example (b)

- Is every number in a list positive?
 (apply and (map < 0 ' (5 6 7 8)))
- This is a nice idea, but will not work

=== context ===

/Applications/PLT/collects/scheme/private/misc.ss:74:7

 Map takes a function and <u>lists</u> for each of its arguments

Simple example (c)

- · Is every number in a list positive?
- Use (lambda (x) (< 0 x)) as the function (apply and (map (lambda (x) (< 0 x)) '(5 6 7 8)))
- This works nicely and gives the right answer
- What we did was to use a general purpose, two-argument comparison function (?<?) to make a narrower one-argument one (0<?)

Simple example (d)

- Here's where curry helps (curry < 0) ≈ (lambda (x) (< 0 x))
- So this does what we want (apply and (map (curry < 0) '(5 6 7 8)))
 - Currying < with 0 actually produces equivalent of: (lambda x (apply < (append '(0) x)))
 - So (curry < 0) takes one or more args, e.g.
 ((curry < 0) 10 20 30) => #t
 ((curry < 0) 10 20 5) => #f

[But '< taking more than 2 args makes example a toy[©]]

A real world example

- I wanted to adapt a Lisp example by Google's <u>Peter Norvig</u> of a simple program that generates random sentences from 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 ③ ⑤
- · Scheme's curry helped solve this!

```
#lang scheme

;;; This is a simple ...

(define grammar

'((S -> (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 )

...

(PP -> (PREP NP))

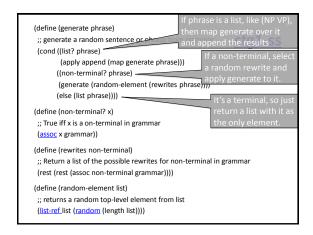
(PP? -> () () () () PP)

))
```

```
cfg1.ss
scheme> scheme
Welcome to MzScheme v4.2.4 ...
                                                  session
> (require "cfq1.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 'NP)
(a worm under every mysterious blue penguin)
> (generate 'NP)
(the book with a large large dog)
```

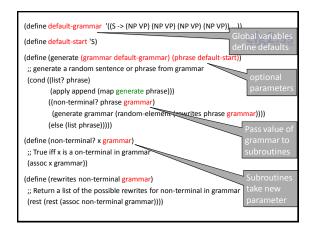
#lang scheme

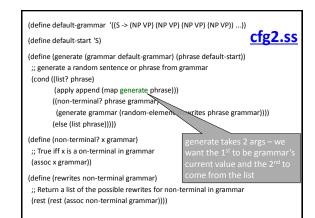
#l



Parameterizing generate

- Let's change the package to not use global variables for grammar
- The generate function will take another parameter for the grammar and also pass it to non-terminal? and rewrites
- While we are at it, we'll make both parameters to generate optional with appropriate defaults





(define default-grammar '((S -> (NP VP) (NP VP) (NP VP) (NP VP)) ...)) cfg2.ss (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)))) (else (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))))

Curried functions

- Curried functions have lots of applications in programming language theory
- The curry operator is also a neat trick in our functional programming toolbox
- You can add them to Python and other languages, if the underlying language has the right support