Principles of Programming Languages, Appendix

Matteo Pradella

February 2015

1. Scheme
2. Haskell
3. Prolog

Libraries (or Modules)

1. The concept of library is pervasive in computer languages
2. basic idea: libs contain procedures, programs, utilities (but also macros) that can be distributed *independently*
3. in many languages they often rely on **namespaces** (e.g. in C++, Java, Common Lisp, ML variants, but not in C, Objective-C, older Schemes, ...)
4. main idea: new **namespace** for the library, **import** of symbols/definitions from other needed libs, **export** of defined symbols (e.g. data, procedures, macros)
5. some languages offer very sophisticated and flexible libraries, e.g. the ML family with **functors** (not in F#)
6. Standard R6RS is the first to introduce this concept in Scheme (but there is a strong tradition of implementation-specific variants, e.g. that of Scheme 48, inspired by ML)

Modules in Racket

1. Like in many Lisps, Racket uses `require` for importing, and `provide` for exporting symbols and their definitions:

```
#lang racket
(provide
    <exported-names>)
(require
    ; if missing, (require racket) is implicit
    <imported-names>)
<implementation>
```

2. The implicit name of the module is its file name (without the type extension)
3. To load and "enter" into a module's namespace, we can also perform `(enter! "<filename>")` at the REPL

lsets: a simple example of a module

1. this is a very simple implementation of a set library with lists
2. main idea: we represent sets as lists, and provide typical set operations, $\cup$, $\cap$, set difference

```
#lang racket
(provide
 simple-set-difference
 simple-intersection
 simple-union
 lset-union
 lset-intersection)
(require
 racket/base) ; we import only the base lib
```

3. (Racket offers a much faster and more complete `racket/set` library)

lsets (cont.)

1. here is a concise (and not very efficient) implementation:

```
(define (simple-set-difference x y) ; y \ x
  (filter (lambda (t) (not (member t x))) y))

(define (simple-intersection x y)
  (filter (lambda (t) (member t x)) y))

(define (simple-union x y)
  (append x (simple-set-difference x y)))
```

2. Note: **member** is not a predicate, because if it finds t, it returns the rest of the list starting from it

lsets (cont.)

1. here are the last operations:

```
(define (lset-union . sets)
  (foldl simple-union (car sets)
                      (cdr sets)))

(define (lset-intersection . sets)
  (foldl simple-intersection (car sets)
                             (cdr sets)))
```

2. to use it: `(require "lset.ss")`

Another module example: redefining standard procedures

1. The require/provide mechanism is flexible: it is possible to rename, append or prepend strings to imported symbols, ...
2. Using **require** with renaming, it is possible also to re-define standard procedures
3. e.g. we first rename + to **old-**+, when we import it:

```
(import
  (rename-in racket/base (+ old-+))
  ...)
```

4. now we can re-define it for concatenating lists and vectors (like in Python, Ruby):

redefinition of +

```
(define +
  (lambda args
    (if (not (null? args))
        (apply
         (cond
           ((string? (car args)) string-append)
           ((list?   (car args)) append)
           ((vector? (car args)) vector-append)
           (else old-+))
         args)
        0))) ; + without arguments
```

redefinition of +: examples

```
(+ 2 3)                          ; => 5
(+ "=2" "+" "3"))                ; => "=2+3"
(+ '(1 2 3) '(4 5 6))            ; => '(1 2 3 4 5 6)
(+ '#(1 2 3) '#(4 5 6))          ; => #(1 2 3 4 5 6)
```

1. this capability is present in many languages (e.g. C++, Ruby)
2. some explicitly forbid it (e.g. Java, ML)

Read

1. In Lisps, the parser is always available through the standard procedure called **read**
2. **read** gets a string from the input (or from a file), parsing (but not evaluating) it
3. e.g.

```
(define my-input (read))
(display (list? my-input))(space)
(display (eval my-input))
```

4. if we write "ciao" (with _ quotes!), we obtain `#f ciao`
5. if we write (+ 1 2), we obtain `#t 3`

Simple I/O

1. a very easy way of working with files: **with-input-from-file**, **with-output-to-file** - parameters are filename and a thunk with the operations to be performed
2. **reads** are used to parse the file and get its contents, while the usual **displays** can be used to put data in the file:

```
(with-output-to-file "temp.txt"
  (lambda ()
    (display "(+ 1 2)")(newline)))

(with-input-from-file "temp.txt"
  (lambda ()
    (let ((v (read)))
      (display (eval v)))))
```

dynamic-wind

1. For a procedure call, the time between the invocation and its return is called its **dynamic extent**
2. We saw that **call/cc** allows reentering a dynamic extent of a procedure after its return
3. There is a procedure, called **dynamic-wind**, that is used to perform operations when entering and/or exiting the dynamic extent
4. its three arguments are the procedures **before**, **thunk**, and **after**, all without arguments
5. **before** is called whenever the dynamic extent of the call to **thunk** is entered; **after** when it is exited
6. useful e.g. for managing files (open/close) used by **thunk**

dynamic-wind (example)

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
     (lambda () (add 'connect)) ; before
     (lambda ()                 ; thunk
       (add (call/cc
             (lambda (c0)
               (set! c c0)
               'talk1))))
     (lambda () (add 'disconnect))) ; after
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))
;=>(connect talk1 disconnect connect talk2 disconnect)
```

dynamic-wind (example)

```
(let ((n 0))
  (call/cc
   (lambda (k)
     (dynamic-wind
      (lambda ()              ; BEFORE
        (set! n (+ n 1))
        (k))
      (lambda ()              ; THUNK
        (set! n (+ n 2)))
      (lambda ()              ; AFTER
        (set! n (+ n 4))))))
  n)   ;; => 1
```

1. in this case **thunk** is not executed, because k is used to escape

Cooperative multitasking (green threads) with **call/cc**

1. Green threads are lightweight threads supported usually by the language
2. this means that they are **not** OS threads, and are lighter
3. very useful e.g. for server-side processing; used e.g. in Erlang, Stackless Python
4. We see here how to implement them using **call/cc** (example taken from Wikipedia)

Cooperative multitasking (cont.)

1. A naive queue for thread scheduling, it holds a list of continuations "waiting to run".

```
(define *queue* '())

(define (empty-queue?)
  (null? *queue*))

(define (enqueue x)
  (set! *queue* (append *queue* (list x))))

(define (dequeue)
  (let ((x (car *queue*)))
    (set! *queue* (cdr *queue*))
    x))
```

Cooperative multitasking (cont.)

1. This starts a new thread running (proc).

```
(define (fork proc)
  (call/cc
   (lambda (k)
     (enqueue k)
     (proc))))
```

2. This yields the processor to another thread, if there is one.

```
(define (yield)
  (call/cc
   (lambda (k)
     (enqueue k)
     ((dequeue)))))
```

Cooperative multitasking (cont.)

1. This terminates the current thread, or the entire program if there are no other threads left.

```
(define (thread-exit)
  (if (empty-queue?)
      (exit)
      ((dequeue))))
```

Cooperative multitasking (cont.)

1. We can try it with this example procedure:

```
(define (do-stuff-n-print str max)
  (lambda ()
    (let loop ((n 0))
      (display str)(display " ")
      (display n)(newline)
      (yield)
      (if (< n max)
    (loop (+ 1 n))
    (thread-exit)))))
```

Cooperative multitasking (cont.)

1. Create two threads, and start them running.

```
(fork (do-stuff-n-print "This is A" 4))
(fork (do-stuff-n-print "This is B" 5))
(thread-exit)
```

Cooperative multitasking (cont.)

1. output:

```
This is A 0
This is B 0
This is A 1
This is B 1
This is A 2
This is B 2
This is A 3
This is B 3
This is B 4
This is B 5
```

Handling **nondeterminism**: McCarthy's Ambiguous Operator

1. **choose**: it is used to choose among a list of *choices*.
2. if, at some point of the computation, the choice is not the right one, one can just **fail**.
3. it is very convenient e.g. to represent **nondeterminism**
4. think about automata: when we have a nondeterministic choice among say `a`, `b`, or `c`, we can just `(choose '(a b c))`
5. main idea: we use **continuations** to store the alternative paths when we **choose**
6. if we fail, we **backtrack**

A Scheme implementation (1)

1. Scheme supports first class continuations, so it is very easy to implement **choose**:

```
(define *paths* '())

(define (choose choices)
  (if (null? choices)
    (fail)
    (call/cc
      (lambda (cc)
        (set! *paths*
          (cons (lambda ()
                  (cc (choose (cdr choices))))
                *paths*))
        (car choices)))))
```

A Scheme implementation (2)

1. and this is **fail**, to manage rollbacks:

```
(define fail #f)
(call/cc
  (lambda (cc)
    (set! fail
      (lambda ()
        (if (null? *paths*)
          (cc '!!failure!!)
          (let ((p1 (car *paths*)))
            (set! *paths* (cdr *paths*))
            (p1)))))))
```

Example

1. a simple example:

```
(define (is-the-sum-of sum)
  (unless (and (>= sum 0)(<= sum 10))
    (error "out of range" sum))
  (let ((x (choose '(0 1 2 3 4 5)))
        (y (choose '(0 1 2 3 4 5))))
    (if (= (+ x y) sum)
      (list x y)
      (fail))))
```

How is Prolog's **cut** implemented?

1. it should be easy to understand it by considering our Scheme implementation of **choose**
2. an implementation of a very basic **cut** is the following:

```
(define (cut)
  (set! *paths* '()))
```

3. so, when we call it, we "forget" all the paths that we saved before
4. in a sense, it is a "point of no return" - either this path succeeds, or we fail

Currying (alio modo)

1. a utility function for currying

```
(define (curry func . curry-args)
  (lambda args
    (apply func (append curry-args args))))
```

2. examples

```
((curry + 1) 2)       ; => 3
((curry + 1 2) 3)     ; => 6
((curry + 1 2) 3 4)   ; => 10
```

Continuation passing style (CPS)

1. If the language does not nativley support continuations, we can build them explicitly: the idea is to use a closure to create the continuation object
2. we can see it with an example, first let us consider a (non tail-)recursive function:

```
(define (rev lst)
  (if (null? lst)
      '()
      (append (rev (cdr lst))
              (list (car lst)))))
```

3. we can make it *tail recursive*, by putting the the operations that are to be performed after the recursive call (i.e. its **continuation**) in a closure, and then passing it (like in our **fold-right-tail**)
4. what does this function, and what is the continuation in this case?

Continuation passing style (cont.)

1. here is the tail-recursive one, with a new parameter holding the continuation k

```
(define (rev-cont lst)
  (define (rev-cont-aux lst k)
    (if (null? lst)
        (k '())
        (let ((continuation
               ;; here is the continuation
               (lambda (x)
                 (k (append x
                            (list (car lst)))))))
          (rev-cont-aux (cdr lst)
                        continuation))))

  (rev-cont-aux lst (lambda (x) x)))
;; i.e. the first continuation is the identity
```

Haskell: An exercise on infinite lists

1. knowing that
2. any :: (a -> Bool) -> [a] -> Bool
3. takeWhile :: (a -> Bool) -> [a] -> [a]
4. what is the meaning of this code?

```
isprime n = not . any (\x -> mod n x == 0) .
                      takeWhile (\x -> x^2 <= n) $
                      primelist
primelist = 2 : [x | x <- [3,5..], isprime x]
```

5. note: any (>0) [3,-3..(-30)] is true; takeWhile (> 0) [3,-3..(-30)] is [3]

The State monad

1. we saw that monads are useful to automatically manage **state**
2. let us define now the monad to do it
3. first of all, we define a type to represent our state:

```
newtype State st a = State (st -> (st, a))
```

4. the idea is having a type that represent a computation having part of the input (and of the output) that represents a state

The State monad

1. here is the actual definition:

```
instance Monad (State state) where
  return x = State (\st -> (st, x))
  State f >>= g = State (\oldstate ->
                          let (newstate, val) = f oldstate
                              State f'        = g val
                          in f' newstate)
```

A first toy example

1. it is the old one, but in another monad

```
esm :: State Int Int
esm = do x <- return 5
         return (x+1)
```

2. what is the result of the following code?

```
let State f = esm
in (f 333)
```

Utilities

1. to use the state monad, it is a good idea to define a few utility functions, for accessing state:

```
getState :: State state state
getState = State (\state -> (state, state))

putState :: state -> State state ()
putState new = State (\_ -> (new, ()))
```

Another toy example

1. variant I

```
esm' :: State Int Int
esm' = do x <- getState
          return (x+1)
```

2. what is the result of the following code?

```
let State f = esm'
in (f 333)
```

Yet another toy example

1. variant II

```
esm'' :: State Int Int
esm'' = do x <- getState
           putState (x+1)
           x <- getState
           return x
```

2. what is the result of the following code?

```
let State f = esm''
in (f 333)
```

Back to trees

1. going back to our "stateful" example on binary trees (i.e. **mapTreeState**), we can revisit it and give another, more elegant and general definition by using our State monad
2. this is the monadic version of **mapTree**:

```
mapTreeM f (Leaf a) = do
  b <- f a
  return (Leaf b)
mapTreeM f (Branch lhs rhs) = do
  lhs' <- mapTreeM f lhs
  rhs' <- mapTreeM f rhs
  return (Branch lhs' rhs')
```

Back to trees (cont.)

1. as far as its type is concerned, we could declare it to be:

```
mapTreeM :: (a -> State state b) -> Tree a ->
            State state (Tree b)
```

2. on the other hand, if we omit the declaration, it is inferred as follows:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
```

3. this is clearly more general, and means that **mapTreeM** could work with *every monad*

Running the state monad

1. to use **mapTreeM**, it is better to define a utility function, to actually *run* the action, by providing an initial state

```
runStateM :: State state a -> state -> a
runStateM (State f) st = snd (f st)
```

2. at last, here is the code for numbering nodes:

```
numberTree :: Tree a -> State Int (Tree (a, Int))
numberTree tree = mapTreeM number tree
    where number v = do
            cur <- getState
            putState (cur+1)
            return (v,cur)
```

Run:

```
testTree = Branch (Branch
                   (Leaf 'a')
                   (Branch
                    (Leaf 'b')
                    (Leaf 'c')))
                  (Branch
                    (Leaf 'd')
                    (Leaf 'e'))

runStateM (numberTree testTree) 1
```

1. we obtain:

```
Branch (Branch (Leaf ('a',1))
                (Branch (Leaf ('b',2))
                        (Leaf ('c',3))))
        (Branch (Leaf ('d',4)) (Leaf ('e',5)))
```

With another monad:

1. we can also use IO:

```
*Main> mapTreeM print testTree
'a'
'b'
'c'
'd'
'e'
```

Another example: imperative GCD

1. we start with a functional GCD:

```
gcdf x y | x == y = x
gcdf x y | x < y = gcdf x (y-x)
gcdf x y = gcdf (x-y) y
```

2. we want to implement in an "imperative way", where variables are memory cells

Imperative GCD (cont.)

1. this example is similar to the example with binary tree, as we can already use the **State** monad defined before
2. first, the **state** is given by two variables à la von Neumann:

```
type ImpState = (Int, Int)
```

3. accessors:

```
getX, getY :: State ImpState Int
getX = State (\(x,y) -> ((x,y), x))
getY = State (\(x,y) -> ((x,y), y))

putX, putY :: Int -> State ImpState ()
putX x' = State (\(x,y) -> ((x',y), ()))
putY y' = State (\(x,y) -> ((x,y'), ()))
```

Imperative GCD (cont.)

1. now the code:

```
gcdST = do { x <- getX; y <- getY;
  (if x == y then return x else
    if x < y
    then do { putY (y-x); gcdST } -- loop!
    else do { putX (x-y); gcdST })}

run_gcd x y = runStateM gcdST (x,y)
```

An advanced example: computations with resources

1. We will consider computations that "consume" resources. First of all, we define the **resource**: `type Resource = Integer`
2. and the **monadic data type**:
   `data R a = R (Resource -> (Resource, Either a (R a)))`
3. Each computation is a function from available resources to remaining resources, coupled with either a result $\in a$ or a **suspended computation** $\in R\ a$, capturing the work done up to the point of exhaustion
4. (**Either** represents **choice**: the data can **either** be `Left a` **or** `Right (R a)`, in this case. It can be seen as a generalization of **Maybe**)

R is a monad:

```
instance Monad R where
    return v = R (\r -> (r, Left v))
```

1. i.e. we just put the value v in the monad as *Left* v

```
R c1 >>= fc2 = R (\r -> case c1 r of
        (r', Left v) -> let R c2 = fc2 v in c2 r'
        ...
```

2. we call $c1$ with resource r. If r is **enough**, we obtain the result *Left* v. Then we give v to $fc2$ and obtain the second R action, i.e. $c2$.
3. the result is given by $c2\ r'$, i.e. we give the **remaining resources** to the **second action**

R is a monad (cont.)

1. if the resources in r are **not enough**:

```
R c1 >>= fc2 = R (\r -> case c1 r of
                  ...
                  (r', Right pc1) -> (r', Right (pc1 >>= fc2)))
```

2. we just chain $fc2$ together with the **suspended computation** $pc1$

Basic helper functions

1. **run** is used to evaluate $R\ \ p$ feeding resource s into it

```
run :: Resource -> R a -> Maybe a
run s (R p) = case (p s) of
      (_, Left v) -> Just v
       _     -> Nothing
```

Basic helper functions (cont.)

1. **step** builds an $R\ a$ which "burns" a resource, if available:

```
step :: a -> R a
step v = c where
    c = R (\r -> if r /= 0
                 then (r-1, Left v)
                 else (r, Right c))
```

2. if $r = 0$ we have to **suspend** the computation as it is $(r, Right\ c)$

Lifts

1. **lift** functions are used to "lift" a generic function in the world of the monad. There are standard lift functions in **Control.Monad**, but we need to build variants which burn resources at each function application

```
lift1 :: (a -> b) -> (R a -> R b)
lift1 f = \ra1 -> do a1 <- ra1 ; step (f a1)
```

2. we extract the value *a*1 from *ra*1, apply *f* to it, and then perform a *step*
3. *lift*2 is the variant where *f* has two arguments:

```
lift2 :: (a -> b -> c) -> (R a -> R b -> R c)
lift2 f = \ra1 ra2 -> do a1 <- ra1
                         a2 <- ra2
                         step (f a1 a2)
```

Show

1. the simplest *show*: we run the computation with just a unit of resources:

```
showR f = case run 1 f of
    Just v  -> "<R: " ++ show v ++ ">"
    Nothing -> "<suspended>"

instance Show a => Show (R a) where
    show = showR
```

Comparisons

```
(==*) :: Ord a => R a -> R a -> R Bool

(==*) = lift2 (==)

(>*) = lift2 (>)
```

1. For example:

```
*Main> (return 4) >* (return 3)
<R: True>
*Main> (return 2) >* (return 3)
<R: False>
```

Then numbers and their operations:

```
instance Num a => Num (R a) where
   (+)         = lift2 (+)
   (-)         = lift2 (-)
   negate      = lift1 negate
   (*)          = lift2 (*)
   abs          = lift1 abs
   signum      = lift1 signum
   fromInteger = return . fromInteger
```

1. in this way, we can operate on numbers **inside the monad**, but for each operation we perform, we **pay a price** (i.e. *step*)

Example: using our monad

1. Now we see R from the point of view of a typical **user** of the monad, with a simple example
2. first we define if-then-else, then the usual recursive factorial:

```
ifR :: R Bool -> R a -> R a -> R a
ifR tst thn els = do  t <- tst
                      if t then thn else els

fact :: R Integer -> R Integer
fact x = ifR (x ==* 0) 1 (x * fact (x - 1))
```

Example runs

```
*Main> fact 4
<suspended>
*Main> fact 0 -- it does not need resources
<R: 1>
*Main> run 100 (fact 10) -- not enough resources
Nothing
*Main> run 1000 (fact 10)
Just 3628800
*Main> run 1000 (fact (-1)) -- all computations end
Nothing
```

1. in practice, thanks to laziness and monads, we built a **domain specific language** for resource-bound computations

Prolog: a classical example: genealogy

1. every book on Prolog has a variant of this example:

```
grandfather(X,Z) :- parent(X,Y), father(Y,Z).
grandmother(X,Z) :- parent(X,Y), mother(Y,Z).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).

father(gino, adamo).
mother(gino, elena).
father(ella, gino).
mother(ella, vita).
father(ugo, gino).
mother(ugo, vita).
```

a classical example (cont.)

1. we can try it with some queries:

```
?- grandfather(X, adamo).
X = ella .

?- grandfather(X, adamo).
X = ella ;
X = ugo.

?- grandmother(ugo, Y).
Y = elena.
```

Another exercise on Push-down Automata

1. define a **deterministic** implementation of PDA
2. optimize it using **cut**, if possible
3. (is it possible to use **cut** for NPDA? Why?)

Deterministic PDA

1. here is the solution:

```
        % acceptance
config(State, _, []) :- final(State), !.

        % standard move
config(State, [Top|Rest], [C|String]) :-
  delta(State, C, Top, NewState, Push), !,
  append(Push, Rest, NewStack),
  config(NewState, NewStack, String).

        % epsilon move
config(State, [Top|Rest], String) :-
  delta(State, epsilon, Top, NewState, Push), !,
  append(Push, Rest, NewStack),
  config(NewState, NewStack, String).
```

A (more efficient) *quicksort* variant

1. we can avoid the append to make it more efficient, by using another parameter as an *accumulator*

```
qsort([X|L],R0,R) :- part(L,X,L1,L2), !,
                     qsort(L2,R0,R1),
                     qsort(L1,[X|R1],R).

qsort([],R,R).
```

2. e.g.

```
?- qsort([4,3,1,13,32,-3,32],[],X).
X = [-3, 1, 3, 4, 13, 32, 32].
```

tracing with SWI-Prolog

1. it is sometimes useful to **trace** a computation, e.g. `trace(qsort).`

```
[debug]  ?- qsort([2,3,1],[],X).
 T Call: (6) qsort([2, 3, 1], [], _G367)
 T Call: (7) qsort([3], [], _G486)
 T Call: (8) qsort([], [], _G486)
 T Exit: (8) qsort([], [], [])
 T Call: (8) qsort([], [3], _G489)
 T Exit: (8) qsort([], [3], [3])
 T Exit: (7) qsort([3], [], [3])
 T Call: (7) qsort([1], [2, 3], _G367)
 T Call: (8) qsort([], [2, 3], _G492)
 T Exit: (8) qsort([], [2, 3], [2, 3])
 T Call: (8) qsort([], [1, 2, 3], _G367)
 T Exit: (8) qsort([], [1, 2, 3], [1, 2, 3])
 T Exit: (7) qsort([1], [2, 3], [1, 2, 3])
 T Exit: (6) qsort([2, 3, 1], [], [1, 2, 3])
X = [1, 2, 3].
```

2. also, old fashioned debugging with `print/1` and `nl/0`.

Higher-order and meta predicates: an example

1. here is an implementation of `map` (usually called `maplist` in the library)

```
map(_, [], []).
map(C, [X|Xs], [Y|Ys]) :- call(C, X, Y), map(C, Xs, Ys).
```

2. e.g. if we define `test(N,R):- R is N*N.`

```
?- map(test,[1,2,3,4],X).
X = [1, 4, 9, 16].
```

Higher-order and meta predicates (cont.)

1. other predicates are used for **adding facts at runtime**:
   1. `asserta(F)` is used to state F as a **first** clause
   2. `assertz(F)` is the same, but as the **last** clause
2. this is also useful to add facts at the REPL, e.g. `?- assertz(test(N,R):-R is N*N).`
3. another peculiar "meta-predicate" is `var`: succeeds when its argument is a variable
   1. CAVEAT: `var(X)` succeeds without binding `X`!

Higher-order and meta predicates (cont.)

1. Moreover, we know that we cannot match $X(Y) = f(3)$, but sometimes we need something analogous
2. we can do it by using the predicate =.. which is used to **decompose** a term
3. e.g. the query f(2,g(4)) =.. X binds X to the list [f, 2, g(4)]
4. so in the previous case we can do f(3) =.. [X,Y]

Infix predicates

1. infix predicates can be defined (almost) like in Haskell
2. e.g.

```
:- op(800, yfx, =>).   % left associative
:- op(900, yfx, &).    % likewise, with lower priority
:- op(600, xfy, ->).   % right associative
:- op(300, xfx, :).    % not associative
:- op(900, fy, \+).    % prefix not
```

3. note the starting :-, because they are **commands**
4. e.g.

```
?- assert(:- op(600, xfy, ->)).
true.
?- (a -> b -> c) =.. X.
X = [->, a, (b->c)].
```

Example: a symbolic differentiator

1. basic rules

```
d(U+V,X,DU+DV) :- !, d(U,X,DU), d(V,X,DV).
d(U-V,X,DU-DV) :- !, d(U,X,DU), d(V,X,DV).
d(U*V,X,DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
d(U^N,X,N*U^N1*DU) :- !, integer(N), N1 is N-1, d(U,X,DU).
d(-U,X,-DU) :- !, d(U,X,DU).
```

2. terminating rules

```
d(X,X,1) :- !.
d(C,_,0) :- atomic(C), !.
```

3. **atomic** holds with atoms and numbers

a differentiator (cont.)

1. terminating rules (cont.)

```
d(sin(X),X,cos(X)) :- !.
d(cos(X),X,-sin(X)) :- !.
d(exp(X),X,exp(X)) :- !.
d(log(X),X,1/X) :- !.
```

2. chain rule

```
d(F_G,X,DF*DG) :- F_G=..[_,G], !, d(F_G,G,DF), d(G,X,DG).
```

3. note that: 1+2/3 =.. X binds X to [+,1,2/3]

a differentiator (cont.)

1. now, let us try it:

```
?- d(2*sin(cos(x+cos(x))), x, V).
V = 0*sin(cos(x+cos(x)))+2* (cos(cos(x+cos(x)))*
          (-sin(x+cos(x))* (1+ -sin(x)))).
```

Legal stuff

1.
2.