1

Giving a Name to a Function

The problem with the use of lambda expressions is that the function specified by a lambda expression has to be re-specified every time it is used.

What is needed is a way of associating a name to a function defined in terms of a list of atoms that specify function arguments and a body that is the function definition itself, without immediately evaluating the list of arguments or the body.

defun -- this special form function returns the name of a function just defined and as a side effect associates the function with the name:

```
(defun name arg-list   s-exprs )
```


©1993 John Mylopoulos

2

Examples

```
-->(defun  abs  (n)  ( if  ( > n 0 ) n ( -n)))
abs
-->  (abs 6)
6
-->  (abs  (- 6)
6
```

A `defun`-ed function can be used like any built-in or predefined function

Another example:

```
-->  (defun fact  (x)
     (cond
       ((= x   1)  1)
       (t (* x  (fact  (- x 1))))))
fact
-->  (fact 5)
120
```


©1993 John Mylopoulos

3

Optional Parameters

Optional parameters for a function can be defined by marking the optional parameters with the special symbol `&optional`. For example

```
--> (defun root (x &optional n)
      (if n
         (expt x (/ 1 n))
         (sqrt x)))
root
--> (root 9)
3.0              ;; n is bound to nil
--> (root 27 3)
?
```

There are a number of other special-purpose parameter types, including `&rest`, which allows the binding of a variable number of arguments to a single parameter. For example, if

```
(defun raise (x &rest numbers)
   (raiseAux x numbers)
```

`(raise (2 3 5))` calls `(raiseAux 2 (3 5))`

©1993 John Mylopoulos

4

Recursion

Lisp uses recursion as a basic control facility in the same way other, more conventional, languages use iteration of one form or another.

A recursive function `f` includes in its definition directly or indirectly forms that use `f`.

For a form evaluation involving a recursive function to eventually terminate, the *size* of the arguments of recursive function calls should be "smaller" than the original call.

For example,

```
(defun fact (x)
    (cond
      ((= x  1)  1)
      (t    (* x (fact (- x 1))))))
```


©1993 John Mylopoulos

5

Writing Recursive Functions

The body of a simple recursive function has the following structure:

```
(cond (simpleCase1 nonRecursiveValue1)
      (simpleCase2 nonRecursiveValue2)
                   .  .  .
      (hardCase1 f1 Ô recursiveCall1)
      (hardCase2 f2 Ô recursiveCall2)
                   .  .  .
   )
```

Where Ô is a composition operator. For example, the body of `fact`:

```
(cond  ((= x 1)  1  )
       (t        (* n (fact (-  n 1)))))
```

Here the composition operator is `*`

©1993 John Mylopoulos

6

Writing Recursive Functions: The Simple Cases

All recursive functions consist of one or more *simple cases* that require no further recursion.

The *simple case* in `fact` is `(= x 1)`.

The *non-recursive value* is `1` (according to the definition of factorial).

Trivial cases are often defined in terms of `atom`, `nil`, `0`, `1`, along with some builtin function

©1993 John Mylopoulos

7

Writing Recursive Functions: Reduction Operators

Hard Cases are dealt through reduction and composition

• Reduce the hard case to a somewhat less hard case.

For example we know how to compute `(fact x)` in terms of `(fact (- x 1))`, or we know that to search for something in a list we have to look at the head and then the rest of the list. Here the reduction operator is subtraction.

More generally, we can try to apply a reduction operator to the value of the parameters, hoping that if repeatedly applied, will eventually turn into a simple case.

For example, in `fact` the reduction operator is to decrement `x` by `1`. When searching a list, the reduction operator could be `rest`

Reduction Operator: `car`, `cdr`, `−1`, `÷2`,

©1993 John Mylopoulos

8

Writing Recursive Functions: Composition Operators

• Apply a composition operator to the value returned by the somewhat less hard case and return the result.

For example, in `fact` the composition operator is to multiply by `x`. When searching a list, the composition operator simply returns the recursive call case, i.e., is the identity function.

Composition Operator: `cons`, `+`, `*`, identity function,...

©1993 John Mylopoulos

9

Types of Recursion: `car` Recursion

Reduction by `first` (`car`): Trivial case is typically an atom.

For example: The **`very-first`** function finds the very first atom of an s-expr

```
very-first of a                is a
very-first of ((b) (a c))      is b
very-first of (() (((v) v)))   is nil
```

```
(defun very-first (x)
   (cond    ((atom x) x)
            (t    (very-first (car x)))))
```

```
very-first of ((b) (a c))
   very-first of (b)
      very-first of b
         returns    b
      returns    b
   returns    b
```


©1993 John Mylopoulos

10

Types of Recursion: `cdr` Recursion

Trivial case is typically the empty list. For example, `append` is a function with two arguments `l1` and `l2`:

```
(append '(a b c) '(d e) )  is  (a b c d e)
(append () '(x y) )        is  (x y)
(append  '((a) b) () )     is  ((a) b)
```

• **Trivial Case:** If `l1` is null, then the value is `l2`

• **Reduction Operator:** Assume `append` can append the rest of `l1`.

For example, consider `append` of `(a b c)` and `(d e)`. Reduce the problem by using `rest` to append `(b c)` and `(d e)`:

• **Composition Operator:** `cons` will do!
`(cons (car l1) *reduced-value*)`
e.g., `(cons 'a '(b c d e))` is `(a b c d e)`

```
(defun  append  (l1 l2)
  (cond ((null l1) l2)
     (t (cons   (first  l1)
           (append  (rest  l1) l2)))))
```


©1993 John Mylopoulos

11

Types of Recursion: Double Recursion with `car/cdr` Reduction

Reduction by `car` and `cdr`. Trivial case is an atom or the empty list. For example, `equal` (discussed earlier) can be defined as follows:

```
(defun mequal(x y)
 (cond
     ((atom x) (cond
                  ((atom y)(eql x y))
                  (t nil)))
     ((atom y)  nil)
     ((mequal (car x) (car y))
             (mequal (cdr x) (cdr y)))
     ( t      nil)  ))
```

The first two `cond` forms are simple cases, the third clause is a conjunction of reduced cases:

```
(defun  mequal  (x y)
  (cond ((atom x)(and (atom y) (eql x y)))
        ((atom y)  nil)
        (t (and  (mequal (car x)  (car y))
           (mequal (cdr x)  (cdr y)) )) ))
```


©1993 John Mylopoulos

12

Types of Recursion: Tail Recursion

The composition function here is the identity function. Functions with tail recursion may also be categorized as using `cdr` recursion, `car` recursion or `cdr/car` recursion. These functions have (potentially) efficient execution.

For example `all-atomp` determines whether each member of a list is an atom:

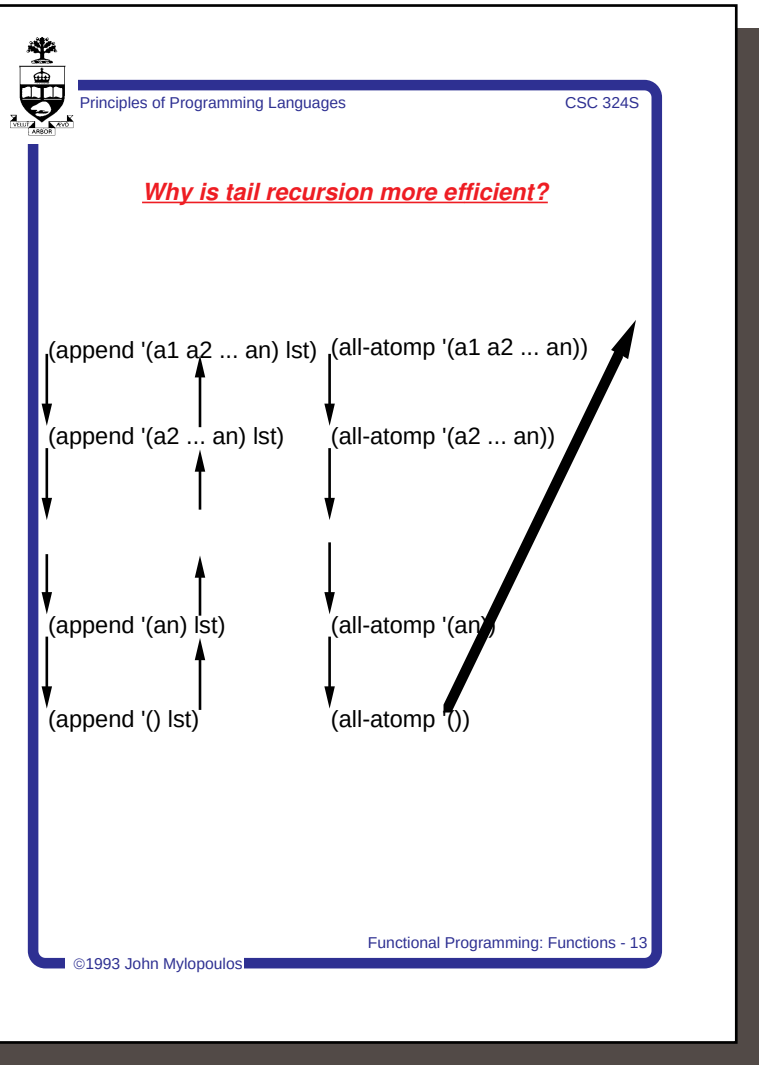
```
(all-atomp  '(a b c))      is    t
(all-atomp  '(a (b c)))    is    nil
(all-atomp   nil)          is    t

(defun     all-atomp  (s)
 (cond  ((null  s)   t)
        ((not (atom (first s)))   nil)
        (t  (all-atomp ( rest s)))))
```

Note that `all-atomp` uses `rest` recursion.

Tail recursion is efficient as the value returned by the innermost `all-atomp` call is simply passed to the outermost `all-atomp` call.

©1993 John Mylopoulos

13


Principles of Programming Languages
CSC 324S
Why is tail recursion more efficient?
(append '(a1 a2 ... an) lst)
(append '(a2 ... an) lst)
(append '(an) lst)
(append '() lst)
(all-atomp '(a1 a2 ... an))
(all-atomp '(a2 ... an))
(all-atomp '(an))
(all-atomp '())
Functional Programming: Functions - 13
©1993 John Mylopoulos