CS 131 Programming Languages

November 10, 2009

Properties of Scheme

Property	**Like**	**Unlike**
Objects are dynamically allocated and never freed	Java, OCaml	C, C++
Types are *latent* not *manifest* (properties of objects)		
Static scoping		
Call by value only		
Good variety of built-in objects, including procedures (incl. *continuations*)	OCaml (not continuations)	
Very simple syntax, a program is straightforwardly represented as data	Prolog	C, C++, Java, OCaml
Tail recursion optimization is required of implementations		
High level, machine-independent arithmetic (no overflow, fractions just work)		

Tail Recursion Optimization

In Scheme, if the last thing a procedure *p* does is call another procedure *q* then *p*'s frame is reclaimed before *q*'s is allocated and reused as *q*'s frame. This is because at that point we know that *p* will return whatever *q* will return.

```
(lambda (x) … some code … (f x y))    ; f is in tail position
(if expr t e)                         ; both t and e are in tail position
(begin e1 e2 e3 … en)                 ; evaluate all expressions and return the value of the last expression
```

Here's a factorial function in Scheme:

```
(define fact
        (lambda (n)
                (if (zero? n)
                        1
                        (* n (fact (- n 1))))))
; this doesn't reuse factorial's frame! the tail call is a multiplication
```

Here's a better factorial function in Scheme, using an accumulator:

```
(define facta                         ; computes n!a
        (lambda (n a)
                (if (zero? n)
                        a
                        (facta (- n 1) (* n a)))))
(define fact
        (lambda (n)
                (facta n 1)))
```

This type of code is very common in Scheme so there's a shorthand notation for it.

```
(define (fact n) … )
```

Scheme has even more sweet stuff that OCaml doesn't. Inside `fact`, we can declare `facta`, an auxiliary function that is meant to be used only inside. What we're doing is called *named let*. A named let both defines and calls the function with given values.

```
(define (fact n0)
        (let facta ((n n0) (a 1))
                (if (zero? n)
                        a
                        (facta (- n 1) (* n a)))))
```

Arithmetic in Scheme is nice. The following number types are available and ***most used***:

inexact	integers	rationals	***real***	***complex***
exact	***integers***	***rationals***	real	complex

Scheme Syntax

Identifiers	`a-zA-Z0-9+-.?*/<=>:$%^&_~`
Comments	`; comment`
Lists (returns result of the expression)	`(a b c d)`
True and False	`#t #f`
Strings	`"string"`
Characters	`#\c`

Unevaluated expression (returns a list)	`Normal quote ('):`
	`'(a b c)`
	`'x = (quote x)`
	``Quaziquote (` and ,)``
	```(+ (3 ,(* 4 x))) ; evaluates 4*x and returns a list (+ 3 4*x)``

Note: in Scheme, empty list counts as *true*.

Internal Program Representation

Internally, a statement like `(a b c d e)` is represented as a linked list with one *pair* per expression. To create a pair, there is a function called `(cons X Y)` and to take it apart there's `(car P)` and `(cdr P)`. To test for empty pair, use `(null? X)` and to test whether something is a pair use `(pair? X)`. It is possible to create an "invalid" list for which last pair doesn't end with a null.

Scheme and Homework 5

```
(define x (list 'list pat '(* 3 x)))
```

The above expression constructs a list containing the arguments `'list`, `pat`, and `'(* 3 x)`. Only `pat` is actually evaluated.

More Built-In Functions

`(eq? a b)`	True if a and b are the same object (pointer comparison)
`(eqv? a b)`	True if a and b have same content (non-recursive comparison)
`(eqv? a b)`	Just like `eqv` but using recursive comparison
`(= a b)`	Compare numbers

When you're writing your own code, try to use `eq` and only use others when necessary, since `eq` is $O(1)$. Scheme allows you to have functions with varying number of arguments:

```
(lambda (x . y) …)     ; 1 or more argument
```

The above function takes a varying number of arguments. First argument is bound to `x` awhile a list with the rest of the arguments is bound to `y`.

```
(lambda (x y . z) …)   ; 2 or more arguments
(lambda x …)           ; 0 or more arguments, all bound in a list to x
```

Scheme can also define the logical `not`, *`and`*:

```
(define (not x) (if x #f #t)
(define not (lambda (x) (if x #f #t)))
(define (and2 a b) (if a b #f))              ; not a standard Scheme 'and'
```

Here's how *`or`* and `and` works in Scheme:

```
(and A B C … Z)                      ; return #f immediately if any expr returns #f, else return Z's value
(or A B C … Z)                       ; return value of first expression that doesn't return #f
```

Let's try to implement `and`:

```
(define (wand . l)
```

```
        (if (null? l)
                #t
                (if (car l)
                        (wand (cdr l))
                        #f)))
```

The above code is wrong and un-elegant. Let's make it more elegant:

```
(define (wand . l)
        (or (null? l)
                (if (null? (cdr l))
                        (car l)
                        (if (car l)
                                (wand (cdr l)))))
```

There above `wand` (our implementation of `and`) is wrong because it doesn't do *short-circuit evaluation*, e.g. it still evaluates next parameter(s) if the first parameter returned `#f`.

Let's try to implement `and` using macros:

```
(define-syntax and
        (syntax-rules ()
                ((and) #t)
                ((and x) x)
                ((and x1 x2 …)
                        (if x1 (and x2 …)))))
```

Let's try to implement `or` using macros:

```
(define-syntax and
        (syntax-rules ()
                ((or) #f)
                ((or x) x)
                ((or x1 x2 …)
                        (let (a x1)
                        (if a a (or x2 …)))))
```

Let's try to do the same using C macros:

```
#define BEGIN  {
#define END    }
#define IF     if (
#define THEN   ) {
#define ELSE   } else {
#define FI     }
```

Now we can write code like this:

```
IF      x == y
THEN    return 3
ELSE    print ("x");
        return 7;
FI
```

Suppose we do something like this:

```
#define f(x, y) return x+y+z;
```

Which instance of `z` are we returning? Whatever was the scope of `z` at the time of macro invocation because macros don't have scope rules (the *problem of macro capture*).

Scheme doesn't have the problem of macro capture because scope is determined at the time of definition of the macro (Scheme macros are *hygienic*).

Look up the book by *Dybvig* on Scheme and read all chapters that mention *continuation*.