CS2040: POPL: Homework #2

Akilesh B

CS13B1042

Functional Programming

PLP CYU:

2)

a) First-class function values and higher-order functions.

b) Extensive polymorphism.

c) List types and operators.

d) Structured function returns.

e) Constructors for structured objects.

f) Garbage collection.

3)

read-eval-print in Scheme just loops, accepting one command at a time, executing it and printing the result.

The three steps at each iteration of the loop are:

a) calling read to read the characters that make up a textual expression from the keyboard input buffer, and construct a data structure to represent it,
b) calling eval to evaluate the expression--intuitively, eval "figures out what the expression means," and "does what it says to do," returning the value of the expression--and
c) calling write to print a textual representation of the resulting from eval, so that the user can see it.

4)

First-class value is one that can be passed as a parameter, returned from a subroutine, or assigned to a variable. It is an entity which supports all the operations generally available to other entities.

11)

The functions eval and apply can be defined as mutually recursive. When passed a number or a string, eval simply returns that number or string. When passed a symbol, it looks that symbol up in the specified environment and returns the value to which it is bound. When passed a list it checks to see whether the first element of the list is one of a small number of symbols that name so-called primitive special forms, built into the language implementation. For each of these special forms (lambda, if, define, set!, quote, etc.) eval provides a direct implementation. For other lists, eval calls itself recursively on each element and then calls apply, passing as arguments the value of the first element (which must be a function) and a list of the values of the remaining elements. Finally, eval returns what apply returned.

12)

In an applicative-order language, all arguments to procedures are evaluated when the procedure is applied.
In contrast, normal-order languages delay evaluation of procedure arguments until the actual argument values are needed.
Delaying evaluation of procedure arguments until the last possible moment is called lazy evaluation.

14)

A function is said to be strict if it is undefined (fails to terminate, or encounters an error) when any of its arguments is undefined. Such a function can safely evaluate all its arguments, so its result will not depend on evaluation order.

15)
Memoization is used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same input occurs again.
Memoization is a programming technique which attempts to increase a function's performance by caching its previously computed results.

17)
A function is said to be a higher-order function if it takes one or more functions as an argument or returns a function as a result.

Eg 1:

```
(define compose
(lambda (f g)
(lambda (x) (f (g x)))))
((compose car cdr) '(1 2 3))
```

Eg 2:

```
(for-each (lambda (a b) (display (* a b)) (newline))
'(2 4 6)
'(3 5 7))
```

Eg 3: (In Haskell)

```
twice function x = (function . function) x

f = (subtract 3)

main = print (twice f 7)
```

18)

Currying is when you break down a function that takes multiple arguments into a series of functions that take part of the arguments.

```
(define (add a b)

(+ a b))
```

When we give (add 3 4), it returns 7.

The above is a function which takes in two arguments, a and b and returns their sum .

Currying the above function.

(define (add a)

(lambda (b)

(+ a b)))

This is a function that takes one argument, a, and returns a function that takes another argument, b, and that function returns their sum.

Currying gives us the ability to pass a partially applied function to a higher-order function.

20)

<u>For side-effect free programming.</u>

a) Initialization of complex structures: The heavy reliance on lists reflects the ease with which functions can build new lists out of the component of the old lists.
b) Summarization: Counting the occurrences of various items or patterns is easy.
c) In-place mutation: Maximize memory usage so that high amounts of data will fit in memory or cache.

<u>Against side-effect free programming:</u>

a) It makes expressions referentially transparent – independent of evaluation order.
b) Equational reasoning in pure functional programming which means equivalence of two expressions at any point implies their equivalence in all times.

21)

Lists are extensively used because they can be built incrementally, without the need to allocate and then modify state as separate operations. They are also important because they have a natural recursive definition and are easily manipulated by operating on their first element and recursively the remainder of the list.

Exercises

10.1

Define is a special form, an exception to the evaluation rule. Usually, an expression represents a procedure invocation, so the general rule is that Scheme first evaluates all the sub-expressions, and then applies the resulting procedure to the resulting argument values. The specialness of special forms is that Scheme doesn't evaluate all the sub-expressions. Instead, each special form has its own particular evaluation rule. For example, when we defined square, no part of the definition was evaluated: not square, not x, and not (* x x). It wouldn't make sense to evaluate (square x) because you can't invoke the square procedure before you define it.

It would be possible to describe special forms using the following model: "Certain procedures want their arguments unevaluated, and Scheme recognizes them. After refraining from evaluating defines' arguments, for example, Scheme invokes the define procedure with those unevaluated arguments." But in fact the designers of Scheme chose to think about it differently. The entire special form that starts with define is just a completely different kind of thing from a procedure call. In Scheme there is no procedure named define. In fact, define is not the name of anything at all:

```
>+
#<PRIMITIVE PROCEDURE +>

> define
ERROR - INVALID CONTEXT FOR KEYWORD DEFINE
```

It is similar to imperative.

10.2

Here are some possibilities:
(a) Functions as first-class objects: functional constants, function types, functions as return values, syntax to call a function returned from another function.
(b) The ability to use statements as expressions, for example to embed an if statement (expression) in a parameter list.
(c) Polymorphic functions
(d) On-the-fly creation of functions, something similar to lambda
(e) Literal constants of arbitrary types constant trees
(f) Garbage collection

10.3

Applicative-Order v/s Normal-Order Evaluation:

1) Applicative-Order Evaluation:
Evaluates function arguments before passing them to a function (Scheme functions use applicative-order evaluation).
Normal-Order Evaluation: Passes arguments unevaluated to a function (Scheme special forms use normal-order evaluation.

2) The obvious difference is short-circuit evaluation in boolean operators. Applicative-order evaluation will evaluate all arguments before evaluating the boolean operator, which could cause unwanted effects if the first condition was guarding the second.

As in, Normal-order evaluation will allow us to short-circuit the evaluation as soon as one of the conditional expressions causes the outcome to become known.

Special forms use delayed evaluation, e.g. else part of a cond expression may not evaluate at all.

Special forms in Scheme (e.g., if and cond) do not use applicative order evaluation.

- Only one of two or more expressions is actually evaluated.
- Evaluation of some expressions is delayed until after another is evaluated (e.g., a then b or c in (if a b c) in Scheme).

The Scheme programming language by R. Kent Dybvig, 4th Edition:

2.2.2:

+ and - (addition and subtraction)

Any combination of integers and rationals results in an integer if possible otherwise a rational is the result, that is:
(+ int int) -> int
(+ int rat) -> int (or rat if int is not possible)
(+ rat int) -> int (or rat if int is not possible)

Any combination of integer or rational and 'double' results in a 'double, that is:
(+ dbl dbl) -> dbl
(+ int dbl) -> dbl
(+ dbl int) -> dbl
(+ rat dbl) -> dbl
(+ dbl rat) -> dbl

For complex numbers the above rules apply to the way the numeric parts of a complex number are returned. That is if the numeric literals are all given as rational, then the returned numeric literals will be given as integers if possible, rational otherwise. The numeric literals will only be returned in 'double' format if the initial numeric literals are expressed in 'double' form.

The operation of multiplication and division follow a similar logic, (possibly the same logic). Where integer answers are possible integer answers are given. If a rational result is possible and none of the operands are in double form then the result will be rational. If a rational answer is not possible then a result in 'double' form is given. If any of the operands are given in 'double' form then result will automatically be given in double form.

3.1.3:

syntax: (let* ((var val) ...) exp_1 exp_2 ...)
returns: the value of the final expression

let* is similar to let except that the expressions val ... are evaluated in sequence from left to right, and each of these expressions is within the scope of the variables

to the left. Use let* when there is a linear dependency among the values or when the order of evaluation is important.

Any let* expression may be converted to a set of nested let expressions. The following definition of let* demonstrates the typical transformation.

```
(let* ((x (* 5.0 5.0))
       (y (- x (* 4.0 4.0))))
  (sqrt y)) ⇒ 3.0

(let ((x 0) (y 1))
  (let* ((x y) (y x))
    (list x y))) ⇒ (1 1)
```

PLCC

Ch #8

8.4

a) f(102)

if 102 > 100 then 102-10 else f(f(102+11))

So, f(102) = 92.

b) f(101)

if 101 >100 then 101-10 else f(f(101+11))

So, f(101) = 91.

c) f(91)

if 91 > 100 then 91-10 else f(f(91+11)).

So, f(91) = f(f(91+11)).

Since, we follow innermost evaluation f(102) will be calculated first then f(f(102))

f(102) = 92.

Therefore, f(91) = f(92).

In a similar manner f(92) will reduce to f(93), f(93) to f(94), f(94) to f(95), f(95) to f(96), f(96) to f(97), f(98) to f(99), f(99) to f(100).

Now, f(100) = if 100 > 100 then 100-10 else f(f(100+11))

f(111) = 101.

f(101) = 91.

Therefore, f(100) = 91.

d) f(90) = f(f(101)) = f(91) = 91.

e) f(89) = f(f(100)) = f(91) = 91.

Ch #10

10.1

a) It checks if a is present in the given list or not. Since list is empty, it will print #f

b) Since a and car'(a b c) match, it will print #t.

c) First, it will check a and car'(c b a), since not true it will then check a and car '(b a) which is again not true, finally it checks for a and car'(a). This matches, hence prints #t.

d) First, it checks a and car'(b), not true. The list then becomes empty implies it prints #f.

e) It prints #f. It checks a and car'(b e d), not true. Then, checks for a and car'(e d), again not true. Then, checks for a and car'(d), not true. Finally, list becomes empty => prints #f.

10.2

a) It prints 5. It checks 5 and '() therefore prints 5.

b) First checks 5 and car '(5 4 3), since 5 is not less than 5 => then checks for 5 and car '(4 3). Now least becomes 4, hence next time checks for 4 and car '(3). Therefore, final answer is 3.

c) First checks 5 and car ‘(4 5 6). Since 4 < 5, least now becomes 4. Next time, checks for 4 and car ‘(5 6). Finally 4 and car’(6). The final answer is 4.

d) First checks 5 and car’(7 3 6 2). Since 5 < 7, least becomes 5. Next time, checks for 5 and car’(3 6 2). Least now becomes 3. It now checks for 3 and car’(6 2). Finally 3 and car’(2) => it prints 2.

10.3

a) Since list is empty, it prints ‘().

b) First, it checks 5 and car’(5 4 3). Since 5 is not less than 5, it now checks for 5 and car’(4 3). Since 4 is less than 5, 4 will be appended to output list. It then checks 5 and car’(3), since 3 < 5, 3 will also be appended to output list.

Therefore ‘(4 3).

c)First, it checks 5 and car’(4 5 6). Since 4 is less than 5, 4 will be appended to output list. It then checks 5 and car’(5 6), since 5 is not less than 5, it then checks for 5 and car’(6). It prints ‘(4).

d) First, checks for 5 and car‘(7 3 6 2). Since 5 is not less than 7, then check for 5 and car’(3 6 2), 3 will be appended to output list. Then check for 5 and car’(6 2) and at last check 5 and car’(2) => output list ‘(3 2).

Finally, answer is ‘(3 2).

8.12)

```
a) let val x =3                    ; x is binded to 3
       in let val y = x + 1        ; y is binded to x+1 (here bound occurrence of x is 3)
             in x + y*y            ; occurrence of x is 3 and y is x+1 = 4
          end
    end                            ; returns 19
```

b) let val x =3 ; x is bind to 3
in let val x = x + 1 ; x is bind to x+1 (here bound occurrence of x is 3)
in x + x*x ; occurrence of x is x+1
end
end ; returns 20

c) let val x =3 ; x is bind to 3
val y = x + 1 ; y is bind to x+1 (here bound occurrence of x is 3)
in x + y*y ; occurrence of x is 3 and y is x+1 = 4
end ; returns 19

d) let val x =
let val x = 3 ; x is bind to 3
in x + 1 ; outer x is bind by x + 1=> the occurrence of x is 3
end
in x + x ; occurrence of x is 4
end ; returns 8

e) let fun sq(x) : int = x*x ; function call x*x
val x=3 ; x is bind to 3
in sq(x) ; occurrence of x is 3
end ; returns 9

f)
```
let val x = 3                    ; x is bind to 3
    fun sq(x) : int = x*x        ; function call x*x
in sq(x)                         ; occurrence of x is 3
end                              ; returns  9
```

g)
```
let fun f(x) = g(1,x)            ; f(x) is bound to g(1,x)
    and g(a,x) = x + a           ; g(x)
    val x = 3                    ; x is bind to 3
in f(x)                          ; occurrence of x is 3
end                              ; returns 4
```