Functional Programming, Scheme

CS152

Chris Pollett

Oct. 29, 2008.

Outline

- More Functional Programming
- Elements of Scheme

More Functional Programming

- Last day, we began talking about functional programming.
- Basically, we defined what it means mathematically to be a function; and we distinguished between the notion of function application and function definition.
- In mathematics, variables always stand for actual values, there is no concept of memory location, and so you can change the value of a variable.
- In a **pure functional programming language**, we would similarly like that there are no variables, only constants, parameters (arguments to functions), and values.
- We build up programs by defining functions of increasing sophistication in terms of previously defined functions.
- The advantage to this approach would be that the language being close to mathematics would have program which would be easier to verify the correctness of.
- Most actual functional programming languages are not completely pure.

What about loops?

- How can you do looping if you have a pure functional programming language?

```
for(int i=0; i <10; i++) {/*do something */}
```

- Seems to require variables; however, we can replace it with recursion:

```
void my_for(int i, int stop, int step) {
    if ( i < stop) { /* do something */
        my_for(i, stop, i+ step)
    }
}
my_for(0, 10, 1);
```

More on Pure Functional Programming

- If one does not have variables and assignment, there is no notion of internal state of a function: The value of a function only depends on its arguments.
- This is called **referential transparency**. I.e., can't use static variables in our functions.
- No assignment and referential transparency, imply the runtime environment can be kept simple: we only need to map names to values -- we don't have to worry about location. This is sometimes called **value semantics**.
- Another feature of pure functional languages is that functions are **first class values**. That is, they can be passed as arguments and returned as values.
- Functions that take functions as arguments or return arguments as values are called **higher-order functions**.

Couldn't we just use C and write programs in a functional way?

- Structured values such as arrays and records cannot be returned values from functions. -- so we end up messing around with pointers and worrying about bounds on things like arrays.
- It is hard to build a value of a structured type directly. Functional language, like ML, provide direct mechanisms for creating recursive data type like binary trees, etc.
- Functions are not first class values. So it is hard to write the function h = comp(f,g) a function which take functions f and g as arguments and outputs their composition.

Scheme

- Scheme was developed in the 1970s at MIT as a variant of LISP for teaching purposes.
- Because the Common LISP standard was only adopted in the 1980s, almost 20+ years after the creation of the first LISP interpreters, and because it was a very big language, Scheme carved out a niche.
- Further, there was a very influential book from MIT: *Structure and Interpretation of Computer Programs* (Abelson, Abelson, Sussman) that used it.

Syntax of Scheme

- All programs in Scheme are expressions, expressions come in two varieties:
  - atoms -- numbers, strings, names, functions, etc
  - lists -- a sequence of expressions separated by spaces and surrounded by parentheses.
- Here is a grammar for expressions:

<expression> ::= <atom> | <list>

<atom> ::= <number> | <string> | <identifier> | <character>| <boolean>

<list> ::= '(' <expression-sequence> ')'

<expression-sequence> ::= <expression> <expression-sequence> | <expression>

Expressions and Evaluation

42 - a number

"hello" - a string

#t - a Boolean true

#\a - the character 'a'

(2.1 2.2 3.1) - a list of numbers

a - an identifier

hello - another identifier

(+ 2 3) - a list consisting of the identifier "+" and two numbers

(* (+ 2 3) (-4 3)) - a list consisting of an identifier and two lists

- To evaluate expressions we use the rules:
  - Constant atoms evaluate to themselves
  - Identifiers are looked up in the current environment and replaced by the value found there
  - A list is evaluated by first evaluating each of its elements. The first element in the list must evaluate to a function. This function is then applied to each of the remaining arguments.

More on Evaluation

- So for example, + in (+ 2 3) evaluates to a procedure for addition, 2 evaluates to 2, 3 evaluates to 3. The procedure for addition is then applied to the rest of the list to get 5.
- Evaluating all of the arguments before applying the function at the root of the expression (in this case +) is called **applicative order evaluation**.
- Consider (2.1 2.2 2.3) . The number 2.1 is not a function, evaluating this list would give an error.
- To prevent a list from being evaluated you can write either (quote 2.1 2.2 2.3) or in shorthand '(2.1 2.2 2.3).
- eval is the opposite operation so (eval (quote (+ 2 3))) yields 5. Technically, in the official standard you need to write this as: (eval (quote (+ 2 3)) (scheme-report-environment 5)) where (scheme-report-environment 5) returns the environment binding provided by the version 5 revision of the ANSI standard.

Conditionals in Scheme

- If statements:

```
(if (= a 0) (display "zero") (display "not zero"))
; displays echoes to current output
; semicolon is used for comments
```

- cond (sorta like switch/case):

```
(cond ((= a 0) (display "zero"))
      ((> a 0) (display "bigger than zero"))
      ((= a -1) (display "minus one"))
      (else (display "less than -1")))
```

- Notice neither if or cond use applicative order evaluation. Instead, the use some kind of **delayed evaluation**.

let

- Scheme has function called let which allows values to be given temporary names within an expression:

  (let ((a 2) (b 3)) (+ a b)) ; evaluates to 5

- The first expression within let is called a **binding list**.

Adding things to the Scheme Environment

- The define function can be used to add new associations between names and values in Scheme:

```
(define a 2)
(define emptylist '())
(define (sum lo hi) ; could write: sum (lambda (lo hi) …
   (if (= lo hi)
       lo
       (+ lo (sum (+ lo 1) hi))) )
```

- Once something has been define can see its value are scheme prompt

➢ (sum 3 5)

12