CSCC24: Principles of Programming Languages Notes

Joshua Concon

University of Toronto Scarborough – Winter 2018

This course is taught by Dr. Albert Yu Cheong Lai. If you find any problems in these notes, feel free to contact me at conconjoshua@gmail.com.

Contents

1 Thursday, January 11, 2017

The purpose of this course is to see the trade-offs between various features in programming languages. This course exists because different programming languages have different features, for example, Java has both class-based OOP and auto-garbage collection while C has neither, but C has union types that Java doesn't have. This means rewriting code into a different language isn't necessarily easy. There may be large semantic differences

1.1 The Root Cause of this Course

A guy named "John Backus" gave a lecture for the acceptance for the Turing Award in 1977. He addressed the question, "Can programming be liberated from the von Neumann style?"

Languages then had been only superficial enhancements to the CPU writing 1 word onto memory at a time i.e:

```
 s := s + a[i]
```

Backus proposed a new direction for programming languages:

- Higher order functions that work on aggregates (a whole list, an array, a dictionary, etc...)
- Combining forms, for example, function composition ($g \circ f$)
- Reasoning by algebra, for example, the associative law for a function
- If you need a state, use coarse-grained state transitions rather than changing only one word at a time. (So passing an old state into a stateless function that does a lot and returns an answer and a new state.)

1.1.1 Higher-order Functions on Aggregates

Note that the notation to apply a function to several parameters is:
(Haskell)

```
    f x y z
```

(Scheme:)

```
(f x y z)
```

So in Haskell:

```
fmap f [x0, x1, ...]
```

will compute

```
[f x0, f x1, ...]
```

And

```
fmap abs [3,-1,4]
```

Computes

```
[3,-1,4]
```

And

```
folder (+) 0 [3,1,4]
```

Computes

```
3+(1+(4+0))
```

Note 2 points:

- ”on aggregates” means to work on a whole list at once (such as an array or some ”container”)
- ”Higher-order functions” means that some parameters are functions, so different combinations makes the language more customizable.

Java and MATLAB have the former but lack the latter.

1.1.2 Combining Forms

An obvious example is function composition $(g \circ f)$.
In Haskell, this is:

```
g . f
```

And in Racket (Scheme) this is:

```
compose g f
```

For example, the following code computes the 1-norm of your vector.

```
foldr (+) 0 . fmap abs
```

There are other combining forms. There is another example in Haskell.

```
    (f &&& g) x = (f x, g x)
```

The point is that you can combine functions to perform compound tasks, and this type of language is not about shorter code (although it has that side effect), but about working with building blocks.

1.1.3 Example Topic: Evaluation Order

You can define your own logical "and" in Scheme

```
    (define (my-and b c) (if b c #f))
    (my-and #f (list-ref '(#t #f #t) 10))
```

The second line fails in Scheme, but if typed in the Haskell version, succeeds.

In most languages, parameters are evaluated before passed into the bodies of functions. In Haskell however, parameters are passed as is. Because of this, in Haskell, many short circuiting operators and control constructs are user-definable, and therefore, very customizable.

1.1.4 Example Topic: Scheme Macros

Scheme offers a macro system for user-defined constructs:

```
    (define-syntax-rule (my-and b c) (if b c #f))
```

Now if we run the following code, it succeeds.

```
    (my-and #f (list-ref '(#t #f #t) 10))
```

The explanation for this is that this is a macro expansion in Scheme, so the parameters are copy-pasted into the macros. This means that there is a downside, for example:

```
    (define-syntax-rule (double x) (+ x x))
    (double (* 3 4))
```

The second line spawns two copies of (* 3 4) and performs redundant work, while Haskell's version does not. The Upside is that Scheme's macro system offers other flexibilities not shown in this lecture.

1.1.5 Dynamic and Static Typing

In Scheme:

```
    (if #f 0 (+ 0 "hello"))
    (if #t 0 (+ 0 "hello"))
```

The first line fails but the second line succeeds. This is because Types are checked dynamically. When running the program, only the code that is actually run is checked.
In Haskell, the following line fails:

```
    if True then 0 else 0 + "hello"
```

The reason for this is because types are checked statically, without running, over all the code. (If this code is compiled, then at compile time, if interpreted, then at load time, etc.) So the error of adding 0 to "hello".

Food for though, Java is both compiled and interpreted.

1.1.6 Parametric Polymorphism

In Haskell, we define:

```
    trio x = [x, x, x]
```

The inferred type is:

```
    a -> [a]
```

This is analogous to Java's

```
    <T> LinkedList<T> trio(<T> x)
```

Note: That the following 2 lines are both legal if we have trio defined

```
    trio 0
    trio "hello"
```

"Parametric" means: Supposed you have defined d of type $a \mapsto [a]$, Then you would need one test to know what it does. Say we test $dTrue$ and the answer has length 2. Then we can deduce that dx returns $[x, x]$ for all x.

The basic explanation for this is that d cannot vary behaviour by types. Haskell allows type-determined behaviour, but the function type will look like:

```
    Foo a => a -> [a]
```

1.1.7 What is "Powerful"? – The Tradeoff

"Macro systems, dynamic typing, ... are powerful." This refers to the flexibility for the implementer or the original author.

"Static typing, parametric polymorphism... are powerful." This refers to the predictability for the user or the maintainer.

Programming is a dialectic class struggle between the user and the implementer. Or between the maintainer and the original author.

2 Thursday, January 18, 2018

2.1 Racket

We won't be using just Scheme, we'll be using Racket which is a version of Scheme. Racket is a platform for implementing and using many languages, and Scheme is on of those that come out of the box.

Racket's version of scheme is somewhat different from the standards with regards to function names, and some features. We will cover Racket, but note that these examples and features may fail for standard Scheme.

2.2 Basic Data Types

```
    #t, #f ;booleans
    42 ;numbers, can be ints, rational, floats, complex
    "hello" ; strings
    #\h ; this is a char of just the letter h
    'Chrome ;this is a symbol
```

Symbols Symbols are user-defined atomic values. You think of a name, put a single quote in front. Symbols are not strings, you can't perform string operations onto them.

2.3 Procedures and Functions

For example:

```
    (sin (/ 0.2 2)) ; sine of 1 over 10
```

2.4 Boolean Operations

```
    (not expr)
    (and expr expr)
    (or expr expr)
    (or) ; gives #f
    (and) ; gives #t
    (boolean? expr) ; tests if you have a boolean
```

2.5 Number operations

```
    +
    -
    *
    /
    max
    min
    -
    <
    >
    <=
    >=
    ; these operations can all take multiple operands
```

```
    number?
    complex?
    real?
    rational?
    ; these functions test if a number is a certain type
```

2.6 Equality

There are 3 types of equalities.

```
    eq?
```

This one is Good for booleans and symbols, uses pointer inequality for aggregates like strings and lists, but has complicated rules for numbers.

```
    eqv?
```

This one has complicated rules for numbers as well, and different from eq? as it treats the floating-points NaN and signed zero differently.

```
    equal?
```

This one is for structural equality for most aggregates. So comparing contents of an aggregate.

2.7 Definitions

You can define functions and constants, recursion is allowed.

```
; A constant
(define my-width/height (/ 4 3))
; a function with 2 parameters
(define (my-log base x)
  (/ (log x) (log base)))
```

2.8 Anonymous Functions

Basically a function without a name, you can either right lambda or use λ in your code.

Example of a function:

```
    (lambda (base x) (/ (log x) (log base)))
```

Example of the same function being used

```
    ((lambda (base x) (/ (log x) (log base))) 128 2)
```

So in this case, base passed in as 128 and x as 2

2.9 Conditionals

If-then-else conditions look like the following

```
  (if test then-expr else-expr)
```

Test can be non-boolean, and this will be treated as true.
Multiple conditions are as such:

```
  (cond
    [(> x y) (sin x)]
    [(< x y) (cos y)]
    [else 0])
```

This is if $x > y$ then $sinx$ else if $x < y$ then $cosy$ else 0. Test results can be non-boolean, which are treated as true. You can obtain the result of a test to return it.

```
  (cond
    [(+ 4 2) => (lambda (x) (* x x))]
    [else 0])
```

This gives 36.

2.9.1 and,or as conditionals

and evaluates all of its operands from left to right and stops as soon as #f operand is read, otherwise the last expression is the answer.

or evaluates all of its operands from left to right and stops as soon as a non #f operand is read, and that becomes the answer, otherwise the answer is # f

2.10 Local bindings

Local definitions for use in just one expression

```
(let ([x expr1]
  [y expr2])
  (+ x y (* 2 x y)))
```

This means to compute $x + y + 2xy$ where $x = expr1$ and $y = expr2$. These 2 expressions cannot see the others variables, only the global ones outside of their scope.

```
(let ([x 3])
  (let ([x (* 3 3)]) ; (* 3 3)
  x))
```

This results in 9 and is not recursive. Let* allows later bindings to see earlier bindings

```
(let* ([x 5]
  [y (+ x 1)]) ; (+ 5 1)
  (+ x y (* 2 x y)))
```

2.10.1 Recursive local bindings

letrec allows more recursive bindings

```
(letrec ([fac
  (lambda (n)
    (if (= n 0) 1 (* n (fac (- n 1)))))]
  [even
    (lambda (n)
      (or (= n 0) (not (odd (- n 1)))))]
  [odd
    (lambda (n)
      (not (even (- n 1))))])
(even (fac 5)))
```

This returns whether or not factorial 5 is even. So true.

2.11 Recommended Code Layout

- Open parentheses then immediately first word
- Procedure definition: Body starts on new line, indented
- Long expression: Parts start on new lines, indented

- Closing parentheses not on new lines

3 Thursday, January 25, 2018

4 Thursday, February 1, 2018

4.1 Scheme (cont'd)

4.1.1 foldl

Consider the problem of summing an entire list and multiplying an entire list. Summing requires us to add up all the elements of the list plus 0 for the first item. Multiplying requires us to multiply up all the elements of the list times 1 for the first item. This is the motivation.

So we define foldl as

```
  (define (foldl binop a lst)
(match lst
[?() a]
[( cons hd tl) (foldl binop (binop a hd) tl )]))
```

So Intuitively,

```
(foldl binop a (list x y z))
```

Looks like

$$(((a + x) + y) + z)$$

where + is where binop is being performed

4.1.2 foldr

Basically in the opposite direction of foldl, so if

```
(foldl binop a (list x y z))
```

Looks like

$$(z + (y + (x + a)))$$

where + is where binop is being performed, then

```
(foldr binop a (list x y z))
```

Looks like

$$(((a + x) + y) + z)$$

where + is where binop is being performed, then

4.1.3 Procedure-Call Stack

Consider the following:

```
(define (f n) (... (f (- n 1)) ...)
(displayln (+ (f 4) (f 1) (f 6)))
```

The Control-flow jumps to into f when it's called and later knows where to return to after the recursive calls. This is done because a stack is used to remember where to return to in recursion, called a **Call Stack**. The Benefit of this is that it supports recursion, but it comes at a price of occupying $\theta(1)$ space while the stack is being used.

4.1.4 Non-Tail Calls and Tail Calls

Non-Tail Calls, are if you still have to do your own processing or computation after getting the results from another function, so basically the results are not returned right away.

For example, for the following function:

```
(define (my-sum lst)
  (match lst [?() 0]
      [( cons hd tl) (+ hd (my-sum tl ))]))
```

Takes $\theta(n)$ space if the list length is n.

A Tail call is the exact opposite, there is no computation after getting the results back from a called function and the function returns the value right away. The complexity of this is $O(1)$ under Tail-Calling optimization in Scheme.

Tail-Calling optimization isn't in every language, Java and Python don't have this.

4.2 Haskell

4.2.1 Expressions and Types

Characters chars are denoted with single quotes

Tuples Not the same as cons

() is a special time, called the unit type, used as a return value for functions that don't return anything

Lists Lists are implemented as such:

```
3 : ( 1 : (4 : [])) = [3,1,4]
```

So a list of length one is an item with an empty list and the colon inbetween separates the items

```
[1] = 1 : []
```

Note that the following list has a type [[*integer*]]

```
[[3,1,4], [10,20], []]
```

So as for now, arbitrary list nesting is not supported, so basically

```
[1, [3]]
```

is not supported (yet).
Note that because of static typing, every item must be the same type, we can't mix integers with floats in the same list.

Strings are a list of chars. The downside of this is that it uses a huge amount of memory, as it's stored as a linked list, and each node and pointer takes up a linear amount of space.

Keyword: Just If you type

```
Just 'C'
```

The variable is either Nothing or Char

Nothing Is the empty type, can be any type

"Left 'C'" Can either be of type Char Bool, Char Int, ... all we know that the left variable is a character.

"Right False" Can either be of type Char bool, Int bool, ... all we know that the right variable is a boolean

anonymous functions Ex.

```
  \x -> x >= 'C'
```

Has the type Char -> Bool with char as the domain and bool as the codomain

4.2.2 Definitions

We can define expressions to variables and vice versa, for example, in the following, we are defining "ten" to be "1+2+3+4" and binding "1+2+3+4" to "ten":

```
ten = 1 + 2 + 3 + 4
```

There is also pattern binding, with tuples, but that will be shown later

Functions can also be defined:

```
square x = x * x
nand a b =  not (a && b)
```

We can also define type signatures for the definitions as such

```
ten, four :: Integer
```

But Haskell is written such that the type signature can be separated from the definition, so you don't have to put them in the same few lines, they just have to be in the same file.

4.2.3 Function Applications

If you insert one parameter to a function that takes 2 parameters, that function will return a function of 1 parameter. This is how Haskell does multiple parameters

5 Thursday, February 8, 2018

5.1 Haskell (cont'd)

5.1.1 Local Definitions For Expressions

```
let x = 4 + 5
     y = 4 - 5
 in x+y+2*x*y
```

Layout Version above. Braced Version below

5.1.2 Local Definitions For Definitions

```
for u v = x + y + 2*x*y
  where
    x = u + v
    y= u - v
```