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:

```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:

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

language=haskell

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:

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

language=haskell

The inferred type is:

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

This is analogous to Java's

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

language=Java

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]
```

language=Haskell

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

3.1 Pairs and Lists

A cons cell is a 2-tuple pair and has the following syntax:

```
cons(x y)
```

Essentially a pair of pointers. Has special support for lists.

```
'() ; an empty list
(list x y z) = (cons x (cons y (cons z '())))
'(42 "hi" Chrome)
; Chrome here will be the symbol 'Chrome
```

For a cons cell, you can use car to access the first field and cdr to access the second field.

3.2 User-Defined Records

```
(struct dim (width height))
```

This creates a new record type with 2 fields

```
(dim 4 7) ; this constructs a vlue of this type
dim? ; this tests for this type
dim-width
dim-height
; these are the field accessors
```

We can also use struct-copy to clone a record while replacing some values:

```
(define d1 (dim 4 7))
(define d2 (struct-copy dim d1 [width 5]))
; d2 is (dim 5 7)
```

3.3 Pattern Matching

You can test for a literal, cons cell, or a record type, can get their content as well.

```
(struct dim (width height))

(define (foo x)
    (match x
        ['() 'nada]
        [(cons b _) b]
        [(dim w h) (* w h)]))
(foo '()) ; returns 'nada
(foo '(1 2 3)) ; returns 1
(foo (dim 4 7)) ; returns 28
```

3.4 Input and Output

We can print with display, printf and displayln

```
(display 5)
(newline)
(displayln 5)
(printf "yes" "price" 5)
```

```
(read-line) ; this reads a lien
(read-string 10) ; this reads up to the upper bound.
;if it reaches the end of the file, it returns eof, which
   you can use eq? or eof-object? to test
; for stderr, eprintf is like printf but goes to stderr
```

3.4.1 ports

Racket has ports, analogous to Java Reader/Writer – behind it can be file, string, network connection, message queue, user-defined, etc.

3.5 Sequencing

If we want to evaluate multiple expressions in the order we specify

```
(begin
    (display ln "Please enter your name")
    (read-line)) ; this returns the last expression

(begin0 expr1 expr2) ; this returns the first expression,
   but the others are still evaluated.

(when (> x 0) expr1 expr2 ...)
; if true, evaluates the exoressions, returns what the
   last one returns, if false, returns #<void>
```

3.6 Mutable Varibles

```
(define v 5)
(define (f x) (+ x v))
(f 0) ; this gives 5
(set! v 6)
```

```
(f 0) ; this gives 6
```

Mutable paris, list, strings, arrays, etc. are also available. Use mutation judiciously, is not that necessary.

3.7 map

Takes in a function and a list and applies the function to every element in that list

```
(map f (list x y z)) = (list (f x) (f y) (f z))
```

3.8 filter

filter takes in a boolean function and a list (A) and returns a list of the items in the list A that satisfy the boolean function

```
(filter number? '(9 "4" 0 "1" "6" 5)) = '(9 0 5)
```

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

```
    foo u v = x + y + 2*x*y
        where -- this where refers to the statement in line 1
            x = u + v
            y= u - v
```

5.1.3 Pattern Matching

Can be done for expressions or function definitions as well as pattern binding.

```
-- expression case
case expr of
    [] -> 0
    42 : xs -> foo xs
    x : xs -> x + foo xs
```

In this example, the expression *expr* would evaluate to 0 if it was an empty list, have foo applied to its tail if it started with 42 as its first index and x plus foo applied to its tail for any other list.

```haskell
-- function definition case examples

mySum [] = 0
mySum (x : xs) = x + mySum xs

nand False _ = True
nand True False = True
nand True True = False
```

```haskell
-- pattern binding case
[a, b, c] = take 3 someList
-- a = take
-- b = 3
-- c = someList
```

5.1.4 Guards

Guards are extra conditions imposed on patterns.

```haskell
-- expression case
case expr of
    [] -> 0
    x : xs | x < 0 -> x + foo xs
           | x > 2 -> x - foo xs
           | True -> x * foo xs

-- definition case
foo [] = 0
foo (x : xs) | x < 0 -> x + foo xs
           | x > 2 -> x - foo xs
           | True -> x * foo xs
```

Instead of True for the edge case, can also use ”otherwise”.

5.1.5 Local Definitions under Patterns and Guards

```
foo :: Either String Integer -> Integer
foo (Left str) | suffix > "albert" = 42
            | otherwise = 24
            where
                suffix = drop 10 str

foo (Right x) | x > 0 = 2*y
             | x < 0 = y
             | otherwise = 0
             where
                y = div 1000 x
```

Note that the first where belongs to the foo where the input is a Left str, the second where belongs to the foo where the input is a Right integer and if $x = 0$, then y will not be calculated.

5.1.6 List Comprehension

```
    [x + y | x <- [10,20,30], x > 10, y <- [4,5] ]
    -- this results in
    --[20+4, 20+5, 30+4, 30+5]
```

We can use pattern matching as well

```
    [x+3 | Just x <= [Just 10, Nothing, Just 30]]
    -- this results in
    -- [10+3, 30+3]
```

There is also a range notation we can use:

```
    [1...5] -- this is the same as [1,2,3,4,5]
```

5.1.7 Algebraic Data Types

```
data MyType = Nada | Duplet Double String | Uno Integer
```

Nada, Duplet, Uno are data constructors. They must start with uppercase letters. They form expressions and patterns.

The following is an example function that takes in ”MyType”:

```
plus1 :: MyType -> MyType
plus1 Nada = Nada
plus1 (Duplet r s) = Duplet (r+1) s
plus1 (Uno i) = Uno (i+1)
```

List, unit, tuple, Maybe, and either are algebraic data types from the standard library.

Recursive definitions are ok too, like the following:

```
data Stack = Button | Push Int Stack
```

5.1.8 Parametric Polymorphism

consider the following function type contract

```
map :: (a -> b) -> [a] -> [b]
```

Here both a and b are type variables. They start with lowercase letters (actual data types are capitalized, like Bool). The user chooses what types to use for a and b, and the implementer cannot choose what type of a and b, and must let their function work for all types of a and b.

Algebraic data types can be parameterized by type variables too, for example:

```
data Either a b = Left a | Right b
-- We can generalize the previous stack example to
data Stack a = Bottom | Push a (Stack a)
```

5.1.9 Type-Class Polymorphism

We notice that comparison functions like

```
    (==)
    (<)
```

cannot use completely general it its polymorphism, they must take in items as input that are two of the same type of class that can be compared.

So how does Haskell pull these off?

Haskell uses a **type class** that declares overloaded operations. So the example from before:

```
    (==)
    (<)
```

must take in two inputs,lets call them $a- > a$, where they are both of type $Eq\ a$, which is a class that lets the two inputs be compared.

However, note that classes are not the same as types. Eq is not a type, $Bool$ is not a subclass.

So if we were to type this out:

```
(==) :: Eq a => a -> a -> Bool
- "Eq a" is a "class constraint"
```

In this example, the user chooses what type to use for a, but that chosen type must be an instance of Eq.

Note that constraints propagate down the dependency chain:

```
(==) :: Eq a => a -> a -> Bool

eq3 :: Eq a => a -> a -> a -> Bool
eq3 x y z = x==y && y==z
```

6 Thursday, February 15, 2018

6.1 Haskell (cont'd)

6.1.1 Constraint Instances

What if you are comparing instances inside of a data structure (let's say, a list for example)?

Well we can do this:

```
    instance Eq a => Eq [a] where
        [] == []       = True
        (x:xs) == (y:ys) = x==y && xs == ys
        _ == _     = False
```

Constraints propagate down the dependency chain, including other instance implementations

6.1.2 User-Defined Class

```
class ADT a where
    tag :: a -> String

instance ADT (Either a b) where
    tag (Left _) = "Left"
    tag (Right _) = "Right"

instance ADT MyType where
    tag Nada = "Nada"
    tag (Duplet _ _) = "Duplet"
    tag (Uno _) = "Uno"
```

Classes in Haskell are similar to Java's Interfaces in the sense that you implement the classes' functions for each instance as well.

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT

-- The "Eq a =>" here means that
-- Every Ord instance is also an EQ instance
-- (Superclass, subclass)
```

For implementers of type and instances, these superclasses must be specified, but for users, they can use Ord without mentioning Eq.

6.1.3 Auto-Generating Instance Implementations

The compiler is willing to write some instance code for you, for select standard classes: (Eq, Ord, Enum (but no fields allowed), Show, and a few others.

```
data MyType = Nada | Duplet Double String | Uno Integer
    deriving (Eq, Ord, Show)
data Browser = FireFox | Chrome | Edge | Safari
    deriving (Eq, Ord, Show)
```

6.1.4 Haskell's Number System

You can't use doubles and integers together for number operators, you have to convert one so that they are the same (Can use fromIntegral to convert an integer into a double).

6.1.5 Functor

```
fmap_List :: (a -> b) - > [a] -> [b]
fmap_List = map

```

```
fmap_Maybe :: (a -> b) -> Maybe a -> Maybe b
fmap_Maybe f Nothing = Nothing
fmap_Maybe f (Just a) = Just (f a)

fmap_Either :: (a->b) -> Either e a -> Either e b
fmap_Either f (Left e) = Left e
fmap_Either f (Right a) = Right (f a)
```

The pattern here is that there is a $f : a \mapsto b$ induces a corresponding $Fa \mapsto Fb$ where F is a parameterized type. There is a class for that.

```
class Functor f where
    fmap :: (a->b) -> f a -> f b
```

So this function generalizes the previous examples above.

Every instance of Functor should satisfy:

```
fmap id xs = xs
fmap g (fmap f xs) = fmap (g . f) xs
```

fmap also has an infix alias of $<\$>$, for example:

```
sin <$> [1,2,3]
```

6.1.6 Applicatives

You now become ambitious. You ask: What if you have a binary operator, and two lists, ...

```
listCross :: (a -> b -> c) -> [a] -> [b] -> [c]

```

```haskell
maybeBoth :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
maybeBoth op (Just a) (Just b) = Just (op a b)
maybeBoth op _ _ = Nothing
```

And what if you have a ternary operator and three lists?

Can you implement

```haskell
ap_List :: [a -> b] -> [a] -> [b]
```

such that, for example:

```haskell
ap_List [f,g] [1,2,3] = [f 1, f 2, f 3, g 1, g 2, g 3]
```

Answer:

```haskell
ap_List = ListCross (\f -> \x -> f x)
```

Equivalently listCross ($)

Now can implement tenary too:

```haskell
listTenary :: (a->b->c->d)->[a]->[b]->[c]->[d]
listTenary tenary as bs cs =
    ((tenary <$> as) 'ap_List' bs) 'ap_List' cs
```

There is a class for this too.

```haskell
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) - > f a - > f b
```

```haskell

-- example instance
instance Applicative Maybe where
    pure a = Just a
    Just f <*> Just a = Just (f a)
    _ <*> _ Nothing
```

Applicative subsumes Functor, so we can implement fmpa as

```haskell
fmap f xs = pure f <*> xs
```

7 Thursday, March 1, 2018

7.1 Haskell (cont'd)

7.1.1 Monads

So far we have been thinking of List, Maybe, Either,... as data structures (ex. containers). Now we must think of them as programs:

- foo :: Maybe Int means: a program that may return a number successfully, or may abort
- foo :: [Int] means: a non-deterministic program that returns different numbers in different parallel universes.
- foo:: Either String Int means: like Maybe, but if it aborts, it uses Left to tell you an error message

Now we re-read Functor and Applicative from this angle

```
fmap abs foo
-- this now means to return the absolute value
-- of what foo returns

(+) <$> foo <*> bar
-- this now means to return the sum of what
-- foo and bar returns
```

But bar does not know what foo returns, or vice versa.

You now become ambitious. Can you combine two programs such that the return value(s) of the 1st is fed to the 2nd so the 2nd can behave independently? Like such:

```
bind:: F a -> (a -> Fb) -> F b
-- so we can have
-- prog1st 'bind' prog2nd?
```

We can think of prog2nd as a callback to prog1st.

Other examples would look like the following:

```
-- bind for Maybe

bind_Maybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bind_Maybe Nothing _ = Nothing
bind_Maybe (Just a) k = k a

-- bind for List

bind_List :: [a] -> (a -> [b]) -> [b]
bind_List [] _ = []
bind_List (a:as) k = k a ++ bind_List as k
```

There is a class for that too.

```
class Applicative f => Monad f where
    return :: a -> f a
    (>>=) :: f a -> (a -> f b) -> f b

-- example instance

instance Monad [] where
    return a = [a]
    as >>= k = concat (map k as)
```

Remark: return and pure should be the same thing. Historically, Monad came first, Applicative came later, thus the redundancy. There is a proposed change to make return an alias of pure.

Monad subsumes Applicative: Can implement (<*>) as

```
fs <*> as = fs >>=
            \f -> as >>=
```

```
                  \a -> return (f a)
```

There are equations for Monad too, such as:

```
return a >>= k = k a
```

There is ”do-notation” so code looks nicer and the computer emits

```
>>= \v ->
```

for you:

```
fs <*> as = do
   f <- fs
   a <- as
   return (f a)
```

7.1.2 Haskell I/O System

Parametrized type ”IO” for all ”I/O” commands. Instance of Monad, Applicative, Functor.

```
foo : : IO Char
-- this means a program that interacts with the outside
   world, then returns a character (or gets stuck forever, or
   throws an exception).

putStrLn :: String IO ()
getLine :: IO String
-- NOT: getLine :: String
```

Do not think about how to extract the string. Use (>>=) to feed it to the next program (callback).

```
main = getLine >>= \s -> putStrLn("It's " ++ s)
-- OR
main = do
    s <- getLine
    putStrLn ("It's " ++ s)
```

7.2 Syntax

7.2.1 Context-Free Grammar (CFG)

A context-free grammar looks like this bunch of rules:

$$E \to E + E$$

$$M \to M \times M$$

$$A \to 0$$

$$A \to (E)$$

$$E \to M$$

$$M \to A$$

$$A \to 1$$

Main idea:

- E, M, A are non-terminal symbols aka variables. When you see them, you apply rules to expand
- $+, \times, 1, 0, (,)$ are terminal symbols. They are the characters you want in your language

7.2.2 Derivation (aka Generation)

Derivation is a finite sequence of applying the rules until all non-terminal symbols are gone. Often aim for a specific final string.

$$
\begin{aligned}
E &\to M \\
&\to M \times M \\
&\to A \times M \\
&\to 1 \times M \\
&\to 1 \times A \\
&\to 1 \times (E) \\
&\to 1 \times (E + E) \\
&\to 1 \times (M + E) \\
&\to 1 \times (A + E) \\
&\to 1 \times (0 + E) \\
&\to 1 \times (0 + M) \\
&\to 1 \times (0 + M \times M) \\
&\to 1 \times (0 + A \times M) \\
&\to 1 \times (0 + 1 \times M) \\
&\to 1 \times (0 + 1 \times A) \\
&\to 1 \times (0 + 1 \times 1)
\end{aligned}
$$

Context-free grammars can support: matching parentheses, unlimited nesting.

7.2.3 Backus-Naur Form (BNF)

Backus-Naur Form is a computerized, practical notation for CFG

- Surround non-terminal symbols by <>; allow multi-letter names
- Merge rules with the same LHS
- (Some versions.) Surround terminal strings by single or double quotes.

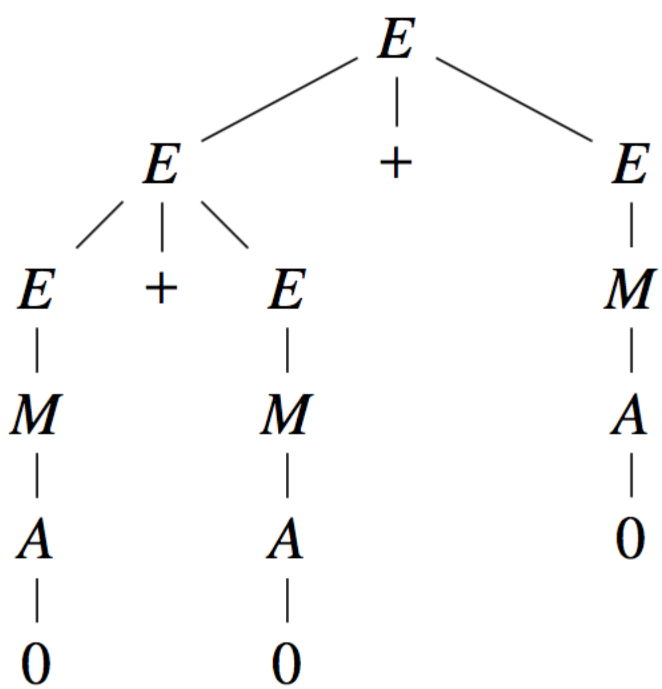
- use ::= for $\rightarrow$

Our example grammar in BNF:

```
<expr> ::= <expr> "+" <expr> | <mul>
<mul> ::= <mul> "*" <mul> | <atom>
<atom> ::= "0" | "1" | "(" <expr> ")"
```

7.2.4 Parse Tree (aka Derivation Tree)

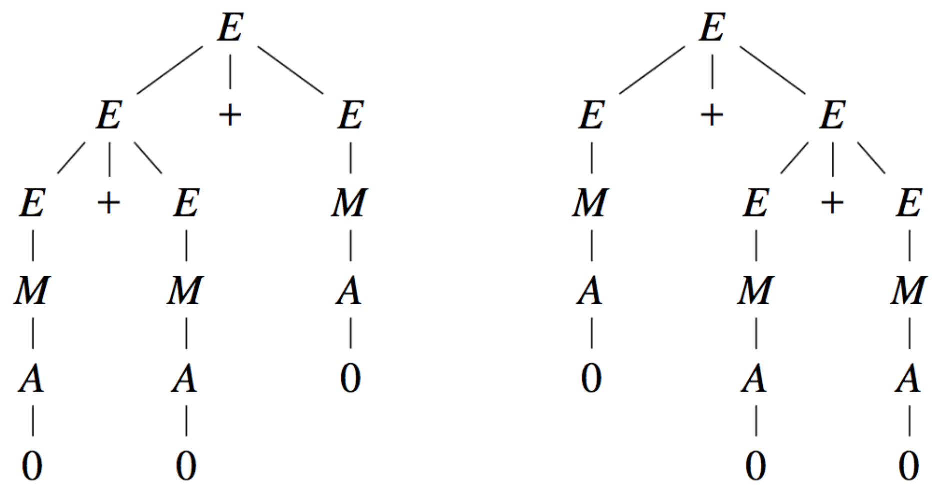
A parse tree aka derivation tree presents a derivation with more structure (tree), less repetition.



This example generates $0 + 0 + 0$

7.2.5 Ambiguous Grammar

Two different trees generate the same $0 + 0 + 0$



If this happens, the grammar is ambiguous.

We try to design unabiguous grammars.

(Bad news: CFG ambiguity is undecidable)

7.2.6 Ubambiguous Grammar Example

An unambiguous grammar that generates the same language as our ambiguous grammar example:

```
<expr> ::= <expr> "+" <expr> | <mul>
<mul> ::= <mul> "*" <mul> | <atom>
<atom> ::= "0" | "1" | "(" <expr> ")"
```

(Bad news: Equivalence of two CFGs is also undecidable)

7.2.7 Left Recursive vs Right Recursive

```
<expr> ::= <expr> "+" <mul>
```

This is an example of a left recursive rule. The recursion is at the beginning (left).

```
<expr> ::= <mul> "+" <expr>
```

This is an example of a right recursive rule. The recursion is at the end (right).

They affect whether infix operators associate to the left or right.

They also affect some parsing algorithms.

7.2.8 Recursive Descent Parsing

Recursive descent parsing is a simple strategy for writing a parser.

- Write a procedure for each rule
- Non-terminals on Right Hand Side become procedure calls, possible recursive calls. (Thus ”recursive descent”. Also ”top-down”.) (Left-recursive grammars need special treatment.)
- Terminal symbols: Consume input and check
- Alternatives require lookahead and/or backtracking

7.2.9 Recursive Descent Parser Example

Example grammar suitable for recursive descent parsing:

```
<sub> ::= <atom> "-" <sub> | <atom>
<atom> ::= "0" | "1" | "(" <sub> ")"
```

Pseudo-code of recursive descent parser:

```
sub:
    try (atom
        read; if not "-" then fail
        sub)
    if that failed: atom

atom:
    read;
    if "1" or "0": return
    if "(" : sub
        read; if not ")" then fail
    else: fail
```

8 Thursday, March 8, 2018

The lectures after this, Albert no longer provides lecture slides, but instead, decides to do everything in haskell files, so I will just be posting the lecture slides here and an explanation.

8.1 ParserLib.hs

Albert provided this really long document called "Pearl.pdf" that explained the concept of parsing, and parsers are essentially built from this one basic parser that "consumes" one character of a string and returns a result based off that input (so think of a parser consuming the c of a $(c : cs)$ string, and returning some result based off that input with the rest of the unparsed string (cs)).

```haskell
-- | Library of parser definition and operations.
module ParserLib where

import Control.Applicative
import Data.Char
```

```haskell
import Data.Functor
import Data.List

newtype Parser a = PsrOf{
    -- | Function from input string to:
    --
    --   * Nothing, if failure (syntax error);
    --   * Just (unconsumed input, answer), if success.
    dePsr :: String -> Maybe (String, a)}

-- Monadic Parsing in Haskell uses [] instead of Maybe to
   support ambiguous
-- grammars and multiple answers.

-- | Use a parser on an input string.
runParser :: Parser a -> String -> Maybe a
runParser (PsrOf p) inp = case p inp of
                            Nothing -> Nothing
                            Just (_, a) -> Just a
                          -- OR: fmap (\(_,a) -> a) (p inp)

-- | Read a character and return. Failure if input is empty.
anyChar :: Parser Char
anyChar = PsrOf p
  where
    p "" = Nothing
    p (c:cs) = Just (cs, c)

-- | Read a character and check against the given character.
char :: Char -> Parser Char
-- char wanted = PsrOf p
--   where
--     p (c:cs) | c == wanted = Just (cs, c)
--     p _ = Nothing
char wanted = satisfy (\c -> c == wanted) -- (== wanted)

-- | Read a character and check against the given predicate.
satisfy :: (Char -> Bool) -> Parser Char
```

```
satisfy pred = PsrOf p
  where
    p (c:cs) | pred c = Just (cs, c)
    p _ = Nothing

-- | Expect the input to be empty.
eof :: Parser ()
eof = PsrOf p
  where
    p "" = Just ("", ())
    p _ = Nothing


-- | Read and check against a given string.
string :: String -> Parser String
string wanted = PsrOf p
  where
    p inp = case stripPrefix wanted inp of
             Nothing -> Nothing
             Just suffix -> Just (suffix, wanted)
           -- Refactor this!

-- But you have to compose smaller parsers to build larger parsers and to return
-- more interesting answers, e.g., abstract syntax trees.
--
-- This is what fmap, pure, <*>, >>= are for. And there are more...

instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap f (PsrOf p) = PsrOf q
                       -- (\inp -> fmap (\(rest, a) -> (rest, f a)) (p inp))
      where
        q inp = case p inp of
                  Nothing -> Nothing
                  Just (rest, a) -> Just (rest, f a)
```

```
                -- fmap (\(rest, a) -> (rest, f a)) (p inp)

instance Applicative Parser where
    -- pure :: a -> Parser a
    pure a = PsrOf (\inp -> Just (inp, a))
    -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
    -- Consider the 1st parser to be stage 1, 2nd parser stage 2.
    PsrOf p1 <*> PsrOf p2 = PsrOf q
      where
        q inp = case p1 inp of
                  Nothing -> Nothing
                  Just (middle, f) ->
                      case p2 middle of
                        Nothing -> Nothing
                        Just (rest, a) -> Just (rest, f a)
                      -- dePsr (fmap f (PsrOf p2)) middle

instance Alternative Parser where
    -- empty :: Parser a
    -- Always fail. The identity for <|> below.
    empty = PsrOf (\_ -> Nothing)
    -- (<|>) :: Parser a -> Parser a -> Parser a
    -- Try the 1st one. If success, done; if failure, do the 2nd one
    PsrOf p1 <|> PsrOf p2 = PsrOf q
      where
        q inp = case p1 inp of
                  j@(Just _) -> j
                  Nothing -> p2 inp
    -- many :: Parser a -> Parser [a]
    -- 0 or more times, maximum munch, collect the answers into a list.
    -- Can use default implementation.

    -- some :: Parser a -> Parser [a]
    -- 1 or more times, maximum munch, collect the answers into a list.
```

```
    -- Can use default implementation.

instance Monad Parser where
    return = pure
    PsrOf p1 >>= k = PsrOf q
      where
        q inp = case p1 inp of
                 Nothing -> Nothing
                 Just (rest, a) -> dePsr (k a) rest

-- | Space or newline or tab.
whitespace :: Parser Char
whitespace = satisfy (\c -> c ‘elem‘ [’\t’, ’\n’, ’ ’])

-- | Consume zero or more whitespaces, maximum munch.
whitespaces :: Parser String
whitespaces = many whitespace

-- | Read and check a terminal string, then skip trailing spaces.
terminal :: String -> Parser String
terminal wanted = string wanted <* whitespaces

-- | Read an integer, then skip trailing spaces.
integer :: Parser Integer
integer = sign <*> (read <$> some (satisfy isDigit)) <* whitespaces
  where
    sign = (char ’-’ *> pure negate) <|> pure id

-- | Read an identifier, then skip trailing spaces. Disallow certain keywords.
identifier :: [String] -> Parser String
identifier keywords = do
    c <- satisfy isAlpha
    cs <- many (satisfy isAlphaNum)
    whitespaces
    let str = c:cs
```

```
    if str `elem` keywords then empty else return str

-- | One or more operands separated by an operator. Apply the operator(s) in a
-- left-associative way.
chainl1 :: Parser a             -- ^ operand parser
        -> Parser (a -> a -> a) -- ^ operator parser
        -> Parser a             -- ^ evaluated answer
chainl1 arg op = do
    a <- arg
    more a
  where
    more x = do
        f <- op
        y <- arg
        more (f x y)
      <|>
        return x

-- | One or more operands separated by an operator. Apply the operator(s) in a
-- right-associative way.
chainr1 :: Parser a             -- ^ operand parser
        -> Parser (a -> a -> a) -- ^ operator parser
        -> Parser a             -- ^ evaluated answer
chainr1 arg op = do
    x <- arg
    ((\f y -> f x y) <$> op <*> chainr1 arg op) <|> return x

-- | Parse a thing that is wrapped between open and close brackets.
between :: Parser open        -- ^ open bracket parser
        -> Parser close       -- ^ close bracket parser
        -> Parser a           -- ^ thing parser
        -> Parser a           -- ^ return the thing parsed
between open close p = open *> p <* close
```

language=haskell

8.1.1 Parser Implementation

So how he sets up Parsers here is that every parser returns a Maybe containing a string of the unconsumed input with a being the result, and if it fails at any point, it returns Nothing. So Parsers are essentially "eating" a string, performing some functions (maybe, success or fail) and then returning the value in a way that it can be further parsed if necessary, or just Nothing if it fails along the way. It's built as a Monad so you can keep applying Parsers to a string easily, and this makes it easier to check for failure.

8.1.2 anyChar

the anyChar function here is a great example here, the Parser consumes one char of the string, and if eats nothing (no more string left to parse), it fails, otherwise, it succeeds and formats the output accordingly

8.1.3 satisfy and char

For char, it uses satisfy which uses a function that if the predicate holds true, then the Parser succeeds and returns what the Parser ate, otherwise, the Parser fails. So char specifically uses a lambda function that checks if it is the argument "wanted".

8.1.4 eof

This one succeeds if the Parser is called on an empty string (nothing left to eat).

8.1.5 empty, many, and some

Just like what the comments say, the empty Parser fails for any input, the many parser runs a parser as many times as it will succeed and populate a list with the results, and will return the list on the first failure, and the some parser does the exact same thing, but there must be at least one success, while the many parser allows for 0 successes.

8.1.6 whitespace, terminal, integer, identifer

whitespace just has the Parser eat one whitespace character, whitespaces eats whitepspace characters with the many parser.

terminal eats all of the whitespaces after a string parser, and identifier eats a string and succeeds if the string is not in the list of provided strings, fails otherwise.

8.1.7 chainl1, chainr1

chainl1 recursively calls the "more" function if it can find more operations and arguments. It coming from the left means that it evaluates it from the left side first (so like this: $(((1 + 2) + 3) + 4) + 5)$.

chainr1 does the exact same thing but on the right side this time $5 + (4 + (3 + (1 + 2)))$.

8.1.8 between

between basically runs its open parser first, then runs the p parser, which result is returned, and then runs the close parser.

9 Thursday, March 15, 2018

9.1 Num.hs

```haskell
module Num where

import ParserLib
import Data.Char
import Control.Applicative

data Expr = N Integer
        | Plus Expr Expr
  deriving (Eq, Show)

interp :: Expr -> Integer
interp (N i) = i
interp (Plus e1 e2) = interp e1 + interp e2


-- expr ::= { integer "+" } integer

-- Left associative below:
-- E.g., 1+2+3 means (1+2)+3

exprParserL :: Parser Expr
exprParserL = do
  i <- integer
  more (N i)
where
  more a = do
      char '+'
      j <- integer
      more (Plus a (N j))
    <|>
      return a

exprParserL2 :: Parser Expr
```

```haskell
exprParserL2 = chainl1 (fmap N integer) op
where
  op = do char '+'
          return Plus

-- Right associative below:
-- E.g., 1+2+3 means 1+(2+3)

exprParser :: Parser Expr
exprParser = do
  i <- integer
  (eof *> return (N i)) <|>
    (do char '+'
        j <- exprParser
        return (Plus (N i) j))

exprParser2 :: Parser Expr
exprParser2 = chainr1 (fmap N integer) op
where
  op = do
      char '+'
      return Plus
```

language=haskell

9.1.1 Purpose

So now that we've done Parsers, we ended up doing an assignment that was about making parsers that turn strings into things like integers and functions, but all in the same format it was given in as a string. Num.hs is essentially about of that into something that actually makes sense.

In this file, we're gonna just be focusing on parsing expressions which are either a Plus of 2 expressions or an integer. We have an interpreter that takes in an integer that represents the number i and just return i and for the Plus case, we'll interpret the two expressions (which will be integers once interpreted), and then we'll return their sum.

9.1.2 Plus function parsers from strings "exprParserL" and "exprParserL2"

If you're reading this and you haven't done the assignment, he also gives an explanation on how to implement a Parser that converts a string into an expression. One implementation that looks like the implementation of chainl1 in ParserLib.hs (because that's essentially what it's trying to do), and the other one utilizes the fact that the "Expr" constructors are also functions. For example, "N" is a function that given an integer "i" returns the data type "(N i)" and "Plus" is a function that given two expressions "a" and "b" will return the data type "(Plus a b)".

9.1.3 expression Parsing from strings

And for "exprParser", it is essentially a Parser for turning strings into our expression type. So $a < | > b$ is the notation for "if Parser a works, return a, otherwise do Parser b" so the two cases are our integer parser that

1. If the string is just an integer "i" (we check this by running eof so that there is no unprocessed input), we return the type "(N i)".

2. If that case does not work or only the integer gets parser'd but the eof fails because there's more stuff, our second case looks for a "+" character and recursively calls itself which would return an integer, it then takes that integer and returns "(Plus (N i) j)". Note that the "i" here comes from the previous case in which eof fails.

9.2 NumVar.hs

```haskell
module NumVar where

import          Data.Map (Map)
import qualified Data.Map as Map
import          Data.Maybe (fromJust)

data Expr = N Integer
        | Var String
        | Plus Expr Expr
  deriving (Eq, Show)

interp :: Map String Integer -> Expr -> Integer
interp env (N i) = i
interp env (Plus e1 e2) = interp env e1 + interp env e2
interp env (Var v) = fromJust (Map.lookup v env)

example = interp (Map.fromList [("x", 5), ("y", 4)])
               (Plus (N 1) (Plus (Var "x") (Var "y")))

interpM :: Map String Integer -> Expr -> Maybe Integer
interpM env (N i) = Just i
interpM env (Plus e1 e2) = fmap (+) (interpM env e1) <*>
    (interpM env e2)
interpM env (Var v) = Map.lookup v env

exampleM = interpM (Map.fromList [("x", 5), ("y", 4)])
                 (Plus (N 1) (Plus (Var "x") (Var "y")))
```

language=haskell

9.2.1 Purpose and interp implementation

This is basically the same as "Num.hs" but introducing environment, which are basically Maps that map strings to integers that are a new argument for the interpreters. So the new case for our interpreter is for a Variable that looks up the string that the var is associated with and just returns whatever it finds.

However, the interp implementation can run into errors, for example, the whole program crashes if it cannot find a variable in the environment (a variable has not been assigned).

9.2.2 interpM implementation

The interpM implementation basically turns its output from an integer to a Maybe Integer, so if the variable cannot be found in the map, it just returns an Nothing, which makes the whole expression evaluate to Nothing instead of crashing.

10 Thursday, March 22, 2018

10.1 NumLet.hs

```
module NumLet where

import           Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import           Data.Maybe (fromJust)

data Expr = N Integer
        | Var String
        | Plus Expr Expr
        | Let [(String, Integer)] Expr -- Unrealistic but
          simpler.
  deriving (Eq, Show)

mainInterp :: Expr -> Integer
mainInterp e = interp Map.empty e

interp :: Map String Integer -> Expr -> Integer
interp env (N i) = i
interp env (Var v) = fromJust (Map.lookup v env)
```

```
interp env (Plus e1 e2) = interp env e1 + interp env e2
interp env (Let bindings e) =
  let env_new = Map.union (Map.fromList bindings) env
  in interp env_new e
  {- In Map.union s1 s2 when a key occurs in both s1 and s2,
     s1's version is
     adopted.

     So here an inner binding shadows an outer binding of the
        same variable
     name --- "shadowing".
  -}

-- (let y=6 in y+4) + (let x=5 in x+1)
example = Plus (Let [("y", 6)] (Plus (Var "y") (N 4)))
               (Let [("x", 5)] (Plus (Var "x") (N 1)))

-- More realistic:
data Expr2 = N2 Integer
           | Var2 String
           | Plus2 Expr2 Expr2
           | Let2 [(String, Expr2)] Expr2
             -- let x=4; y=x+1 in x+y
  deriving (Eq, Show)

{-
Semantics choice:

* Scope: Like let* in Scheme:

     let x=2+3; y=x+4 in x+y

 the y=x+4 has access to x=5.

     let x=y+1; y=x+1 in ...

 x=y+1 refers to y outside, not y=x+1.

```

```
* Evaluation order:

    let x=2+3; y=x+4 in x+y

 Evaluate 2+3, then evaluate x+4, then x+y --- call by value (CBV), aka eager
 evaluation.

 So the environment just needs to map variables to numbers.

Overall, it is as though:

  let x=2+3; y=x+4 in x+y
= let x=2+3 in let y=x+4 in x+y
-}

interp2 :: Map String Integer -> Expr2 -> Integer
interp2 env (N2 i) = i
interp2 env (Var2 v) = case Map.lookup v env of
Just a -> a
Nothing -> error (v ++ " is not found")
interp2 env (Plus2 e1 e2) = interp2 env e1 + interp2 env e2
interp2 env (Let2 [] body) = interp2 env body
interp2 env (Let2 ((v,rhs) : defs) body) =
  let a = interp2 env rhs
      new_env = Map.insert v a env
  in interp2 new_env (Let2 defs body)

mainInterp2 :: Expr2 -> Integer
mainInterp2 = interp2 Map.empty

-- let x=2+3; y=x+4 in x+y
example2 = Let2 [ ("x", Plus2 (N2 2) (N2 3))
                , ("y", Plus2 (Var2 "x") (N2 4))
                ]
                (Plus2 (Var2 "x") (Var2 "y"))

{- let x=2; y=x; z=x+y in ...
```

```

(v,rhs) =  ("x", N2 2) :
defs =  ("y", Var2 "x") : ("z", Plus2 (Var2 "x") (Var2 "y"))
    : []

-}
```

language=haskell

10.1.1 Let implementation

So here, we have essentially what we saw from the previous implementation but now we have a Let type that consists of a list of string integer pairings, the string being the name of the variable and the integer corresponds to its value.

So as you can see in the ”interp” interpreter, when it runs into a Let case, the List of (String, Integer) is conveniently turned into a Map with ”Map.fromList” and we use ”Map.union” to combine our current environment with this Map with the bindings (union will be favouring our new Map for collisions).

So basically that implementation is great if all of your bindings are already calculated for you, but what if your bindings are expressions that rely on previous bindings to be calculated?

So for this new case, we create another datatype ”Expr2” that is exactly like ”Expr” but the bindings are ”(String, Expr2)” pairings. In this example, we want our later bindings to have access to our earlier bindings.

So in our interpreter for this datatype (”interp2”), we do everything in the exact same way again, but for Let, we just interpret the body if the list is empty and if it isn’t, we take the first binding in the list, add it as a variable into a new environment variable, and then recursively evaluate another Let operation with the rest of the variables under the new environment.

10.2 NumLambda.hs

```haskell
module NumLambda where

import           Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import           Debug.Trace

data Expr = N Integer
          | Var String
          | Plus Expr Expr
          | Mul Expr Expr
          | IsZero Expr Expr Expr
          | Lambda String Expr
          | App Expr Expr
    deriving (Eq, Show)

{-
Semantics of App f e: "call by value" (CBV), aka "eager
   evaluation":

1. Evaluate f until it is a value. Expect it to be a lambda
   (later closure).
2. Evaluate e until it is a value.
3. Plug and chug. (Implemented by extending the environment,
   as said.)
-}

-- The type of possible answers from the interpreter.
data Value = VN Integer
           | VClosure (Map String Value) String Expr
             -- Cannot be simply "VFunc String Expr", see below.
    deriving (Eq, Show)

{-
Lambda is evaluated to VClosure. VClosure needs to remember
   an environment
because:
```

```

Suppose your lambda is \y->x+y. What do you use for x?!

Terminology: y is a "bound variable", x is a "free variable".

Bound variable (of an expression): You can see where the
   variable is bound
(introduced or declared or defined).

Free variable (of an expression): You can't, it probably
   comes from the outside.

A free variable like x is probably introduced from an outer
   context:

    (\x -> ..... (\y -> x + y) ....) 10

When the interpreter runs into \y->x+y, and if it already
   knows x=10 from
processing the outer context, it needs to attach "x=10" to
   \y->x+y so it doesn't
forget.

Why not just substitute? Answer: Much slower, and has its own
   subtle problems
--- look up "variable capture" for how easy it is to do it
   wrong. This also
holds for how we implement App: We won't substitute, we will
   just extend the
environment.

Terminology: The combination "\y->x+y, oh BTW x=10 there" is
   a "closure".

A closure is a record or data structure that stores an
   expression (usually a
lambda, but generally can be any expression) together with the
environment/bindings for all of its free variables.
```

```
Fun fact: When the Javascript, Java, and C++ finally added lambda to their
languages, they finally realized how difficult this business of free variables
and closures is! They also have an extra issue:

Since "x" is a mutable variable in their languages, you get to ask: Does it mean
the value of x at the time of creating the lambda, or does it mean a reference
to x?

Java: value at the time of lambda creation ("final variable").
Javascript: reference.
C++: extra syntax for you to choose.
-}

mainInterp = interp Map.empty

interp :: Map String Value -> Expr -> Value
interp env (N i) = VN i
interp env (Var v) = case Map.lookup v env of
  Just val -> val
  Nothing -> error (v ++ " not found")
interp env (Plus e1 e2) = case (interp env e1, interp env e2) of
  (VN i, VN j) -> VN (i + j)
  _ -> error "wrong type in Plus"
interp env (Mul e1 e2) = case (interp env e1, interp env e2) of
  (VN i, VN j) -> VN (i * j)
  _ -> error "wrong type in Mul"
interp env (IsZero test e1 e2) = case interp env test of
  VN 0 -> interp env e1
  VN _ -> interp env e2
  _ -> error "wrong type in IsZero"
interp env (Lambda v body) = VClosure env v body
```

```
interp env (App f e) = case interp env f of
  VClosure fEnv v body ->
      let eVal = interp env e
          bEnv = Map.insert v eVal fEnv -- fEnv, not env
      in interp bEnv body
  _ -> error "wrong type in App"

-- (\x -> (\x -> \f -> f x) 10000 (\y -> x + y)) 10
example1 = App (Lambda "x"
                 (App (App (Lambda "x" (Lambda "f" (App (Var "f") (Var "x"))))
                           (N 10000))
                      (Lambda "y" (Plus (Var "x") (Var "y")))))
              (N 10)

{-
Question: For bEnv, what if we used the following instead?

   bEnv = Map.insert v eVal env

Try with example1.

This was basically the bug made in early implementations of Lisp. Some people
still regard this bug as a feature: "dynamic scoping".

Dynamic scoping: a variable name refers to whichever thing has that name at the
time/place of evaluation.

Lexical/Static scoping: a variable name refers to the innermost outer binding in the code.
-}
```

10.2.1 Implementation

So this implementation is similar to the previous once except it has Multiplication, and a datatype that returns "e1" if "test" evaluates to 0 and returns "e2" otherwise. There is also a lambda data type that takes in a string and an expression.

There's another bit of a switch up here as well, instead of outputting just an integer, the interpreter outputs a "Value" datatype, which could either be an integer, or a "VClosure", which contains an environment, a string, and an expression.

There is also another data type called the "App", with 2 Expressions, the first one is meant to be a function (so after it is interpeted, a VClosure) and the second expression is the input to that function.

10.2.2 Lambda and App interpreted

So when a Lambda is interpreted, it returns a Vclosure with its environment, a string that will be the binding of its input variable (lets call it, "v" for example), and the body of the function.

Now, what happens after this? Well all we can do with a VClosure after this is if the lambda was the first expression of an "App". If we take a look at what happens when an "App" is interpreted, the first expression is interpreted, and if it is a VClosure, then the second input is evaluated, lets call this interpreted second expression "x". So now "x" is inserted into a Map under the key of "v" and then the body of the VClosure is evaluated.

This makes sense, as in Haskell, this is how you pass in input to a lambda function, the expression after the lambda function is evaluated and then put in as input for the lambda function.

10.3 NumRec.hs

```haskell
module NumRec where

import           Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import           Debug.Trace

data Expr = N Integer
          | Var String
          | Op2 BinOp Expr Expr
          | IfZero Expr Expr Expr
          | App Expr Expr
          | Rec String String Expr Expr
  deriving (Eq, Show)

{- Rec is recursive function binding and use.

     let f x = (... may call f ...) in body

 is represented by

     Rec "f" "x" (... may call f ...) body
-}

data BinOp = Plus | Minus | Times deriving (Eq, Show)

binop Plus = (+)
binop Minus = (-)
binop Times = (*)

{-
Semantics of App f e: "call by value" (CBV), aka "eager
   evaluation":

1. Evaluate f until it is a value. Expect it to be a closure.
2. Evaluate e until it is a value.
```

```
3. Plug and chug. (Implemented by extending the environment, as said.)

-}

-- The type of possible answers from the interpreter.
data Value = VN Integer
        | VClosure (Map String Value) String Expr
  deriving (Eq, Show)

mainInterp = interp Map.empty

interp :: Map String Value -> Expr -> Value
interp env (N i) = VN i
interp env (Var v) = case Map.lookup v env of
Just val -> val
Nothing -> error (v ++ " not found")
interp env (Op2 op e1 e2) = case (interp env e1, interp env e2) of
(VN i, VN j) -> VN (binop op i j)
_ -> error "wrong type in Op2"
interp env (IfZero e e0 e1) = case interp env e of
VN 0 -> interp env e0
VN _ -> interp env e1
_ -> error "wrong type in IsZero"
interp env (App f e) = case interp env f of
VClosure fEnv v body ->
    let eVal = interp env e
        bEnv = Map.insert v eVal fEnv
    in interp bEnv body
_ -> error "wrong type in App"
interp env (Rec f v fbody body) =
  let new_env = Map.insert f (VClosure new_env v fbody) env
  in interp new_env body

{-
new_env is a cyclic data structure now. I use Haskell lazy evaluation to set it
```

```
up.

In other languages, use a mutable cell:

Introduce new mutable cell "self".
self := garbage.
new_env := Map.insert f self env
self := VClosure new_env v body
-}

{-
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
-}

fibbody = (IfZero (Var "n")
         (N 0)
         ((IfZero (Op2 Minus (Var "n") (N 1))
            (N 1)
            (Op2 Plus (App (Var "fib") (Op2 Minus (Var "n") (N
               1)))
                     (App (Var "fib") (Op2 Minus (Var "n") (N
                        2)))))))

fib n = mainInterp (Rec "fib" "n" fibbody
                  (App (Var "fib") (N n)))
```

10.3.1 Recursive implementation

So in this example, we have a the same datatype as before, except now the binary operator has been generalized and we have a ”Rec” datatype that takes in 2 strings and 2 expressions. The first string being the name of the function, the second string is the name of the function’s input variable, the first expression is the body of the function and the last expression is the rest of the expression that we need to evaluate.

So when the interpreter reaches the "Rec" datatype, it stores a VClosure of its environment, input variable, and the function body, all under the function name in the map and then evaluates the body.

Remember how (App f e) first evaluates "f" first before evaluating "e"? Well in our lambda example, "f" is just a lambda that evaluates into a (VClosure), but in this recursive example, "f" is just a variable of the function name, and in our implementation of the interpreter, the interpreter just looks for what is the value of the variable name in the map, in this case, if we previously store the function in the map, then the variable would be interpreted into a VClosure, just as the lambda case, and all of the steps would follow from there. This is how function calls would work, if the interpreter finds a function name, it would return the VClosure of that function, and it would do the same thing as a lambda would do from there.

11 Thursday, March 29, 2018

11.1 SelfApp.hs, SelfApp.ss

11.1.1 SelfApp.hs

```
module SelfApp where

import NumLambda

-- \f -> \n -> if n=0 then 1
--             else n * (f f) (n-1)
fac_proto = Lambda "f" (
           Lambda "n" (
             IsZero (Var "n")
               (N 1)
               (Mul
                  (Var "n")
                  (App (App (Var "f") (Var "f"))
                             (Plus (Var "n") (N (-1)))))))
```

```

-- fac n = fac_proto fac_proto n
fac n = mainInterp (App (App fac_proto fac_proto) (N n))

{-
Here is a trace:

   fac_proto fac_proto 2
->
   (\f -> \n -> if n=0 then 1 else n * (f f) (n-1)) fac_proto 2
->
   (\n -> if n=0 then 1 else n * (f f) (n-1)) 2 where f =
      fac_proto = \f ...
->
   if 2=0 then 1 else 2 * (f f) (2-1) where f = fac_proto = ...
->
   2 * (f f) (2-1)  where f = fac_proto = ...
->
   2 * (\f -> \n -> ...) fac_proto (2-1)
->
   2 * (\n -> if n=0 then 1 else n * (fac_proto fac_proto)
      (n-1)) (2-1)
->
   2 * (\n -> if n=0 then 1 else n * (fac_proto fac_proto)
      (n-1)) 1
->
   2 * if 1=0 then 1 else 1 * (fac_proto fac_proto) (1-1)
->
   2 * 1 * fac_proto fac_proto (1-1)
->
   2 * 1 * (\n -> if n=0 then 1 else n * fac_proto fac_proto
      (n-1)) (1-1)
->
   2 * 1 * (\n -> if n=0 then 1 else n * fac_proto fac_proto
      (n-1)) 0
->
   2 * 1 * if 0=0 then 1 else 0 * ...
->
```

```
  2 * 1 * 1
-}

-- \f -> \n -> if n=0 then 0
--              else if n-1=0 then 1
--              else f f (n-1) + f f (n-2)
fib_proto = Lambda "f" (
            Lambda "n" (
              IsZero (Var "n")
                (N 0)
                (IsZero (Plus (Var "n") (N (-1)))
                   (N 1)
                   (Plus (App (App (Var "f") (Var "f"))
                              (Plus (Var "n") (N (-1))))
                         (App (App (Var "f") (Var "f"))
                              (Plus (Var "n") (N (-2))))
                   )
                )
              )
            )

fib n = mainInterp (App (App fib_proto fib_proto) (N n))

-- The theme can be represented by this "diagonal" or self-application
-- combinator, aka "Delta": \x -> x x
diag = Lambda "x" (App (Var "x") (Var "x"))
-- Recall diagonalization proofs from C63...

{-
So why do languages support recursion directly? Why not just make you write
code like this?

Answer: Language design tries to let you express your intention directly rather
than encoding/simulating your intention as something else. Your code is then
```

```
more readable.
-}
```

11.1.2 SelfApp.ss

```
#lang racket

(define (fac_proto f n)
(if (equal? n 0)
    1
    (* n (f f (- n 1)))))

(define (fac n)
(fac_proto fac_proto n))
```

11.1.3 Explanation to Self Application

These two pieces of code do essentially the exact same thing, the first is just in Haskell and the second is in Scheme. But this is just more obvious in the first example.

Think about the factorial function that uses recursion, it's simple enough, it just calls itself with it's name to apply itself recursively. But what if you want to make a recursive lambda? Since the lambda doesn't have a name, how do you call it? So the solution to this is to pass in the function into itself as one of it's arguments, and since you can refer to it's arguments by name, you can refer to the function by name, and this is how lambda recursion works. In this example, the lambda stores itself as "f", and calls itself with the function "f".

11.2 NumStack.hs

```haskell
module NumStack where

import Control.Applicative

data Expr = N Integer
        | Plus Expr Expr
  deriving (Eq, Show)

interp :: Expr -> Integer
interp (N i) = i
interp (Plus e1 e2) = interp e1 + interp e2

{-
Below, an implementation that avoids non-tail recursion. It uses an explicit
custom stack instead. This is closer to a real, low-level implementation:

* tail recursion/call = loop, goto

* stack = stack
-}

data Frame = TODO Expr | Add Integer
  deriving Show

eval :: [Frame] -> Expr -> Integer
eval stack (N i) = exec stack i
eval stack (Plus d1 d2) = eval (TODO d2 : stack) d1

exec :: [Frame] -> Integer -> Integer
exec [] i = i
exec (TODO d : stack) i = eval (Add i : stack) d
exec (Add i0 : stack) i = exec stack (i0 + i)

interp2 :: Expr -> Integer
interp2 = eval []
```

```

{- Here is a trace:

interp2 (Plus (Plus (N 1) (N 2))
             (Plus (N 3) (N 4)))
= eval [] (Plus (Plus (N 1) (N 2))
             (Plus (N 3) (N 4)))
= eval [TODO (Plus (N 3) (N 4))] (Plus (N 1) (N 2)) (*)
= eval [TODO (N 2), TODO (Plus (N 3) (N 4))] (N 1)
= exec [TODO (N 2), TODO (Plus (N 3) (N 4))] 1
= eval [Add 1, TODO (Plus (N 3) (N 4))] (N 2)
= exec [Add 1, TODO (Plus (N 3) (N 4))] 2
= exec [TODO (Plus (N 3) (N 4))] (1+2)           (**)
= eval [Add (1+2)] (Plus (N 3) (N 4))
= ...
= exec [Add (1+2)] (3+4)
= exec [] ((1+2)+(3+4))
= ((1+2)+(3+4))
-}

{-
Invariant:
forall e, stack: eval stack e = exec stack (interp e)

Corollary:
 interp2 e
= eval [] e
= exec [] (interp e)
= interp e

Induction proof: [strong] induction on e.

Base case: e = N i
  WTP: eval stack (N i) = exec stack (interp (N i))

    eval stack (N i)
  = exec stack i
  = exec stack (interp (N i))
```

```

    LHS
  = eval stack (N i)
  = exec stack i

    RHS
  = exec stack (interp (N i))
  = exec stack i

  So LHS = RHS.

Induction step: e = Plus d1 d2
Induction hypothesis:

  forall s: eval s d1 = exec s (interp d1)
  forall s: eval s d2 = exec s (interp d2)

  WTP: eval stack (Plus d1 d2) = exec stack (interp (Plus d1
     d2))

    LHS
  = eval stack (Plus d1 d2)
  = eval (TODO d2 : stack) d1
  = exec (TODO d2 : stack) (interp d1)
  = eval (Add (interp d1) : stack) d2
  = exec (Add (interp d1) : stack) (interp d2)
  = exec stack (interp d1 + interp d2)

    RHS
  = exec stack (interp (Plus d1 d2))
  = exec stack (interp d1 + interp d2)

  So LHS = RHS as wanted.
-}

{-
registers: r0, ep (Expr pointer), stack pointer (implicitly
   used by push and pop)
```

```

label eval:
  if ep points to N i:
      r0 := i
      goto exec
  if ep points to Plus d1 d2:
      push (TODO d2)
      ep := d1
      goto eval

label exec:
  if stack is empty:
      goto exit
  pop
  if that was TODO d:
      push (Add r0)
      ep := d
      goto eval
  if that was Add i0:
      r0 := i0 + r0
      goto exec

label exit:
  the answer is in r0
  so return r0
-}

{-
That stack reminds you what to do next after you finish the
    current small task.

Some people call it the "continuation stack" because it
    represents what/where you
need to continue.
-}
```

11.2.1 Explanation

So this is an implementation of non-tail recursion on a function that only takes in integers and the addition of integers. So the interpreter here just evaluates Expressions until they are integers.

So there are two types of frames that can be in the stack, one is TODO, which is for expressions that have not been evaluated yet, and ADD, which is for numbers that have been computed, but the rest of the equation that belongs with this number has not been computed yet.

In this case, that would be when (Plus e1 e2) is being evaluated. Lets say we evaluate e1 first, we do so and place it on the stack as (Add i) and then evaluate e2. Once they are both evaluated, we add them together and continue to clear items off the stack.

We also have two functions that call each other, one is eval, which evaluates expressions, and exec, which place expressions and integers from the stack to be worked on.

11.2.2 The Trace Example

Here's an explanation of the trace if you don't get it. I'll be refering to the section of the data type where the expressions are evaluated as "the table".

```
-- the entire expression is being interpreted
= interp2 (Plus (Plus (N 1) (N 2))
                (Plus (N 3) (N 4)))
-- eval is called, the stack is empty, and the entire expression is on the table
= eval [] (Plus (Plus (N 1) (N 2))
                (Plus (N 3) (N 4)))
-- the second part of the expression is put on the stack as a TODO, the first part of the expression is on the table
= eval [TODO (Plus (N 3) (N 4))] (Plus (N 1) (N 2))
-- the second part of THAT expression is put on the stack as a TODO, the first part of the original expression is on the table
= eval [TODO (N 2), TODO (Plus (N 3) (N 4))] (N 1)
```

```
-- the expression on the table is evaluated to one
= exec [TODO (N 2), TODO (Plus (N 3) (N 4))] 1
-- the evaluated expression is added to the stack as an Add,
   the next item is removed off the stack to be evaluated
= eval [Add 1, TODO (Plus (N 3) (N 4))] (N 2)
-- this item is evaluated to 2
= exec [Add 1, TODO (Plus (N 3) (N 4))] 2
-- the add is taken off the stack and is added to the value
   on the table
= exec [TODO (Plus (N 3) (N 4))] (1+2)
-- swap what is on the table with what is on the stack
= eval [Add (1+2)] (Plus (N 3) (N 4))
-- do previous steps again
= ...
-- once that expression has been evaluated, go back to the
   stack
= exec [Add (1+2)] (3+4)
-- add the two values together from the stack
= exec [] ((1+2)+(3+4))
-- stack is now empty release the value
= ((1+2)+(3+4))
```

11.2.3 The rest of the file

So the rest of the file is just the proof of the invariant that these functions do in fact, work. So essentially a proof of Correctness. After that there's just a more lower level version of the algorithm, that is using pointers and registers. It looks like assembly.