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

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

```
1    (define (foldl binop a lst)
2   (match lst
3   [?() a]
4   [(cons hd tl) (foldl binop (binop a hd) tl)]))
```

So Intuitively,

```
1   (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

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

Looks like

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

where + is where binop is being performed, then

```
1   (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.

```
-- function definition case examples

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

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

```
-- 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.

```
-- 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]

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

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

such that, for example:

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

Answer:

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

Equivalently listCross ($)

Now can implement tenary too:

```
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.

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

-- 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

```
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

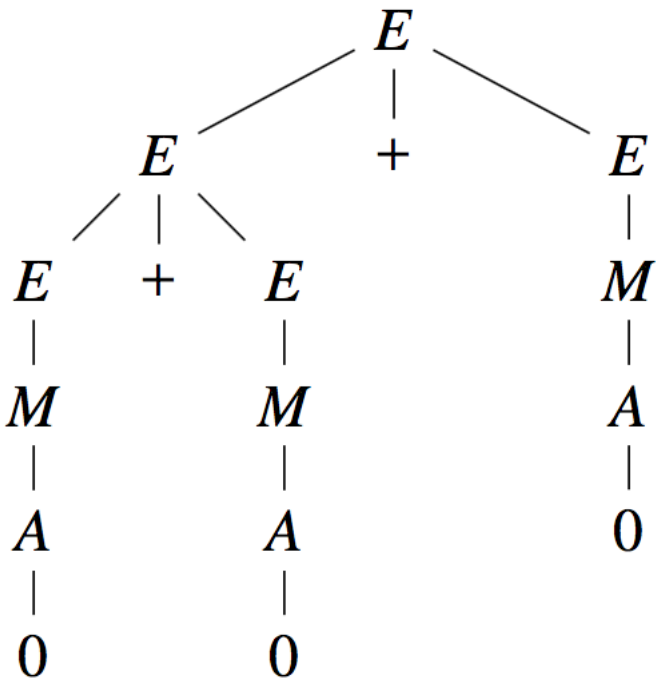
- Surround non-terminal symbols by <>; allow multi-letter names
- Merge rules with the same LHS
- (Some versions.) Surround terminal strings by single or double quotes.
- use ::= for $\rightarrow$

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)

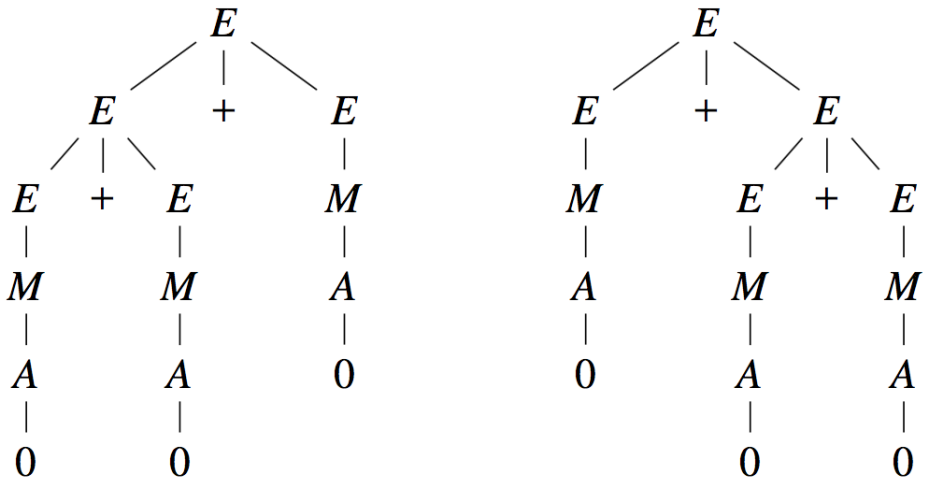
A parse tree aka derivation tree presents a derivation with more structure (tree), less repetition.



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
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
```

```
29    where
30      p "" = Nothing
31      p (c:cs) = Just (cs, c)
32
33  -- | Read a character and check against the given character.
34  char :: Char -> Parser Char
35  -- char wanted = PsrOf p
36  --   where
37  --     p (c:cs) | c == wanted = Just (cs, c)
38  --     p _ = Nothing
39  char wanted = satisfy (\c -> c == wanted)   -- (== wanted)
40
41  -- | Read a character and check against the given predicate.
42  satisfy :: (Char -> Bool) -> Parser Char
43  satisfy pred = PsrOf p
44    where
45      p (c:cs) | pred c = Just (cs, c)
46      p _ = Nothing
47
48  -- | Expect the input to be empty.
49  eof :: Parser ()
50  eof = PsrOf p
51    where
52      p "" = Just ("", ())
53      p _ = Nothing
54
55
56  -- | Read and check against a given string.
57  string :: String -> Parser String
58  string wanted = PsrOf p
59    where
60      p inp = case stripPrefix wanted inp of
61                Nothing -> Nothing
62                Just suffix -> Just (suffix, wanted)
63              -- Refactor this!
64
65  -- But you have to compose smaller parsers to build larger parsers and to return
66  -- more interesting answers, e.g., abstract syntax trees.
67  --
68  -- This is what fmap, pure, <*>, >>= are for.  And there are more...
69
70  instance Functor Parser where
71      -- fmap :: (a -> b) -> Parser a -> Parser b
72      fmap f (PsrOf p) = PsrOf q
73                              -- (\inp -> fmap (\(rest, a) -> (rest, f a)) (p inp))
```

```
74        where
75          q inp = case p inp of
76                    Nothing -> Nothing
77                    Just (rest, a) -> Just (rest, f a)
78                  -- fmap (\(rest, a) -> (rest, f a)) (p inp)
79
80  instance Applicative Parser where
81      -- pure :: a -> Parser a
82      pure a = PsrOf (\inp -> Just (inp, a))
83      -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
84      -- Consider the 1st parser to be stage 1, 2nd parser stage 2.
85      PsrOf p1 <*> PsrOf p2 = PsrOf q
86        where
87          q inp = case p1 inp of
88                    Nothing -> Nothing
89                    Just (middle, f) ->
90                        case p2 middle of
91                          Nothing -> Nothing
92                          Just (rest, a) -> Just (rest, f a)
93                        -- dePsr (fmap f (PsrOf p2)) middle
94
95  instance Alternative Parser where
96      -- empty :: Parser a
97      -- Always fail. The identity for <|> below.
98      empty = PsrOf (\_ -> Nothing)
99      -- (<|>) :: Parser a -> Parser a -> Parser a
100     -- Try the 1st one. If success, done; if failure, do the 2nd one
101     PsrOf p1 <|> PsrOf p2 = PsrOf q
102       where
103         q inp = case p1 inp of
104                   j@(Just _) -> j
105                   Nothing -> p2 inp
106     -- many :: Parser a -> Parser [a]
107     -- 0 or more times, maximum munch, collect the answers into a list.
108     -- Can use default implementation.
109
110     -- some :: Parser a -> Parser [a]
111     -- 1 or more times, maximum munch, collect the answers into a list.
112     -- Can use default implementation.
113
114 instance Monad Parser where
115     return = pure
116     PsrOf p1 >>= k = PsrOf q
117       where
118         q inp = case p1 inp of
```

```
119                     Nothing -> Nothing
120                     Just (rest, a) -> dePsr (k a) rest
121
122 -- | Space or newline or tab.
123 whitespace :: Parser Char
124 whitespace = satisfy (\c -> c `elem` ['\t', '\n', ' '])
125
126 -- | Consume zero or more whitespaces, maximum munch.
127 whitespaces :: Parser String
128 whitespaces = many whitespace
129
130 -- | Read and check a terminal string, then skip trailing spaces.
131 terminal :: String -> Parser String
132 terminal wanted = string wanted <* whitespaces
133
134 -- | Read an integer, then skip trailing spaces.
135 integer :: Parser Integer
136 integer = sign <*> (read <$> some (satisfy isDigit)) <* whitespaces
137   where
138     sign = (char '-' *> pure negate) <|> pure id
139
140 -- | Read an identifier, then skip trailing spaces. Disallow certain keywords.
141 identifier :: [String] -> Parser String
142 identifier keywords = do
143     c <- satisfy isAlpha
144     cs <- many (satisfy isAlphaNum)
145     whitespaces
146     let str = c:cs
147     if str `elem` keywords then empty else return str
148
149 -- | One or more operands separated by an operator. Apply the operator(s) in a
150 -- left-associative way.
151 chainl1 :: Parser a                 -- ^ operand parser
152         -> Parser (a -> a -> a)     -- ^ operator parser
153         -> Parser a                 -- ^ evaluated answer
154 chainl1 arg op = do
155     a <- arg
156     more a
157   where
158     more x = do
159         f <- op
160         y <- arg
161         more (f x y)
162       <|>
163         return x
```

```
164
165 -- | One or more operands separated by an operator. Apply the operator(s) in a
166 -- right-associative way.
167 chainr1 :: Parser a                -- ^ operand parser
168         -> Parser (a -> a -> a)   -- ^ operator parser
169         -> Parser a               -- ^ evaluated answer
170 chainr1 arg op = do
171     x <- arg
172     ((\f y -> f x y) <$> op <*> chainr1 arg op) <|> return x
173
174 -- | Parse a thing that is wrapped between open and close brackets.
175 between :: Parser open          -- ^ open bracket parser
176         -> Parser close         -- ^ close bracket parser
177         -> Parser a             -- ^ thing parser
178         -> Parser a             -- ^ return the thing parsed
179 between open close p = open *> p <* close
```

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.