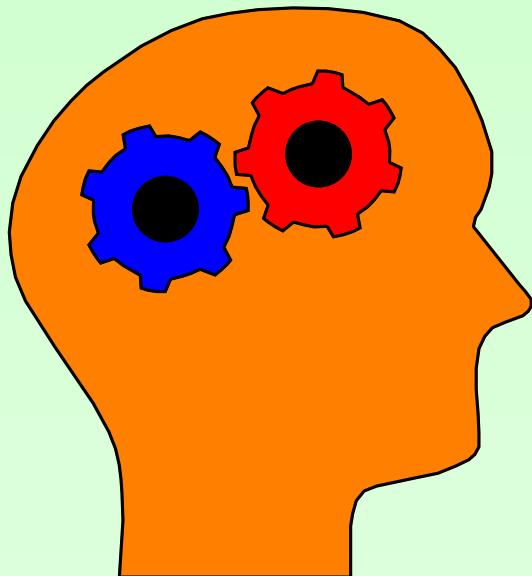




CS2104: Programming Languages Concepts

Lecture 1 : Overview



*“Language Concepts to Support
Programming and Abstraction”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Course Objectives

- cover key concepts in programming languages
- knowledge of language paradigms
- support for software *abstraction, reuse* and *safety*
- improving your programming skills
- make learning a new programming language easier
- highlight successful/advanced languages

Course Outline

- Lecture Topics (12 weeks)
 - Key programming concepts (C, Java, Haskell, OCaml)
 - Values, Types, Functions and Recursion (Haskell)
 - Higher-Order Programming+ Lambda Calculus
 - Type Classes, Monads and Parser Generators (Haskell)
 - Imperative Programming Concepts (with OCaml)
 - OOP Concepts (with Java + Scala)
 - Dynamic Languages (with Python)
 - Logic Programmg (with Prolog)
 - Constraint Programming (with CLP)

Administrative Matters

- IVLE for forum/lecture notes/exercises/submissions
- 2-hour Tutorial/Labs
 - (i) tutorial questions
 - (ii) lab exercises (5 assignments)
- course CA/exams breakdown
 - tutorial participation 5%
 - lab/project assignments 35%
 - term test 15%
 - final examination 45%

Online Reading Textbooks

- lecture slides/notes via IVLE
- Reading Materials (mostly online):
 - ocaml.org
 - caml.inria.fr
 - www.haskell.org
 - www.python.org
 - www.scala-lang.org
 - www.swi-prolog.org
 - Free PL books :
 - <http://www.freebookcentre.net/Language/langCategory.html>
 - <http://www.e-booksdirectory.com/programming.php>

Optional Textbooks

- Concepts of Programming Languages
Robert W Sebesta
- Concepts in Programming Languages
John W Mitchell
- Concepts, Techniques and Models of Computer Programming
Peter Van Roy and Self Haridi (NUS online library)

Lab Assignment/Homework

- Lab assignments in different successful and advanced languages (OCaml, **Scala**, Haskell, Prolog, Python).
- Reinforce concepts taught in class.
- Programming is a skill. It requires lots of practice.
- Pre-requisite to passing course
Do Homework seriously → Pass Course

Why Study Concepts of PLs?

- Inside any successful software system is a good PL

Emacs : *Elisp*

Word, PPT : *VBScript*

Quake : *QuakeC*

Facebook : *FBML, FBJS, Hack (in HHVM)*

Twitter : *Ruby on Rails/Scala*

Also: *Latex, XML, SQL, PS/PDF*

Benefits of Good PL Features

- Readability
- Extensibility.
- Modifiability.
- Reusability.
- Correctness.
- Easy Debugging



What Drives the Development of PL?

- Novel ways of expressing computation
- Better execution model (e.g. dataflow)
- Tackle complex problems (with simpler solution)
- Proof of Concept
- Puristic viewpoint
- Better Reliability
- Domain-Specificity



History of Programming Languages

- Assembly (early 1950s)
- Fortran (late 1950s)
- Lisp (1958)
- Algol (1960s)
- Cobol (1960s)
- **Prolog** (1972)
- C (1973 – birth of Unix)
- Ada (1970s – defense)
- SQL (late 1970s)
- C++ (1985)
- ML (1980), **OCaml** (early 1990), **Haskell** (1987)
- Java (1995)
- Perl, **Python**, Javascript, PHP, VB (1990s)
- **Scala** (first released in 2003)
- C# (2000)
- Go (2009)



Lambda Calculus (1930s)

Programming Paradigms

- Imperative Programming
- *Functional Programming*
- Logic Programming
- Object-Oriented Programming
- Constraint Programming
- Event-Driven Programming (not covered)
- Aspect-Oriented Programming (not covered)

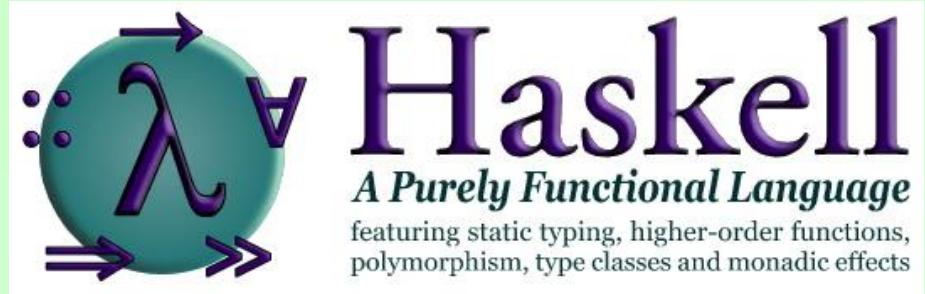
Future of Programming?

What's the future of programming? The answer lies in functional languages

The screenshot shows a news article from TechRepublic. At the top, there's a yellow banner for Scoot airline with the text "DISCOVER OUR DEALS TO OVER 60 CITIES!" and a "Book now" button. Below the banner are social media sharing buttons for comments (4), Facebook, Twitter, LinkedIn, and Email. The main headline reads "What's the future of programming? The answer lies in functional languages". The author is listed as Nick Heath, with a profile picture, and the article was published in Software on October 23, 2017, at 3:35 AM PST. The text of the article states: "Simon Peyton Jones describes functional programming languages like Haskell as a proving ground where programmers can test new ideas." A large photo of Simon Peyton Jones, a man with short grey hair wearing a striped shirt, is centered below the text.

<https://www.techrepublic.com/article/whats-the-future-of-programming-the-answer-lies-in-functional-languages/>

Advanced Language - Haskell



- Strongly-typed with polymorphism
- Higher-order functions
- Pure and Lazy Language.
- Algebraic data types + records
- Exceptions
- Type classes, Monads, Arrows, etc
- Advantages : concise, abstract, pure
- Why use Haskell ?

Cool, clean
and Pure

Hello World in Haskell

```
putStrLn "Hello World!"
```



pure function with type `[Char] -> IO()`

Compilation:

```
ghc -o hello hello.hs
```

Execution:

```
./hello
```

Increment Method (in Haskell)

```
inc :: Int -> Int  
inc x = x+1
```

```
\ x -> x+1
```

(+1)



Example - Haskell Program

- Finite and infinite lists.

a type variable

```
data List a = Nil | Cons a (List a)
```

```
finite_list = Cons 1 (Cons 2 (Cons 3 Nil))
```

finite list of three elements

```
infint n = Cons n (infint (n+1))
```

infinite list starting from [n,n+1,n+2,...]



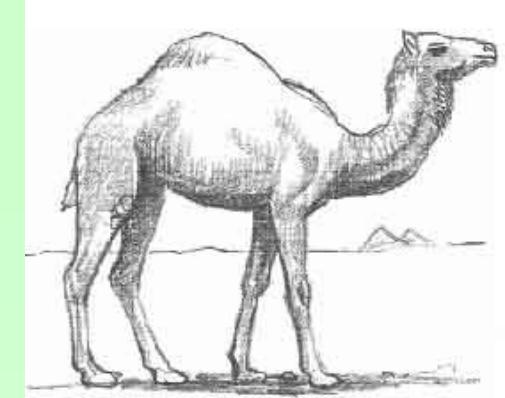
Type System – Lightweight Analysis

- Abstract description of code + genericity
- Compile-time analysis that is tractable
- Guarantees absence of some bad behaviors
- Issues – expressivity, soundness, completeness, inference?
- How to use type system to figure errors.
- Why? 

detect bugs early

Versatile Language - OCaml

- Rich data structures (algebraic data types, records, polymorphism, variants, GADT).
- Typeful higher-order functional and object-oriented language.
- Support for stateful imperative programming.
- Powerful module system.
- Advantages : versatile, abstract, easy reuse
- Why OCaml?



**Powerful, Versatile
and Practical**

Examples (OCaml)

- Hello World

```
print_endline "Hello, World!"
```

- Increment method:

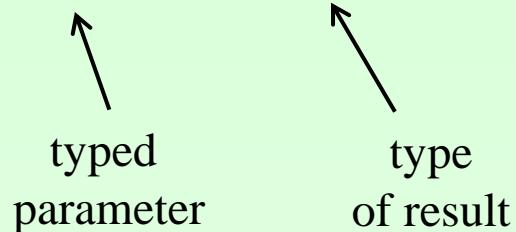
```
fun x -> x+1 end
```

or

```
let inc x = x+1
```

or

```
let inc (x:int) : int = x+1
```



Example – OCaml Program

- Tree Data Structure.

generic tree with type variable ‘a

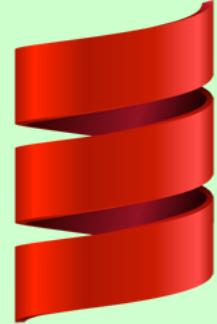
```
type 'a tree = Empty | Node of ('a * 'a tree * 'a tree)
```

type inferred : ('a tree) → int

```
let rec height t =
  match t with
  | Empty -> 0
  | Node(val,lt,rt) -> 1+max(height lt,height rt)
```

**Pattern
Matching**

Scala Programming Language



- stands for “scalable language” – building from reuseable components
- multi-paradigm language
- runs on standard Java and .NET platforms
- interoperates with all Java libraries
- Why study Scala?

concise, Java-compliant

Hello World in Scala

```
object HelloWorld extends App {  
    println("Hello, World!")  
}
```

Compilation:

```
scalac HelloWorld.scala
```

Execution:

```
scala HelloWorld
```

Increment Method

```
object XXX extends App {  
    def inc (x:int) : int = x+1  
}
```

```
(x:Int) => x+1
```

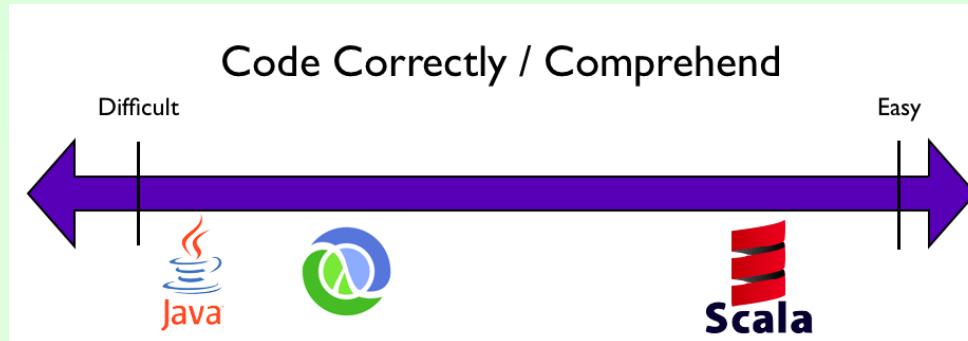
```
new Function1[Int, Int] {  
    def apply(x: Int): Int = x + 1  
}
```



Scala Classes

- Support Java-style classes with Class Parameters, Explicit Overriding + Dynamic Dispatches only

```
class Point(xc: Int, yc: Int) {  
    var x: Int = xc  
    var y: Int = yc  
    def move(dx: Int, dy: Int) {  
        x = x + dx  
        y = y + dy  
    }  
    override def toString(): String  
        = "(" + x + ", " + y + ")";  
}
```



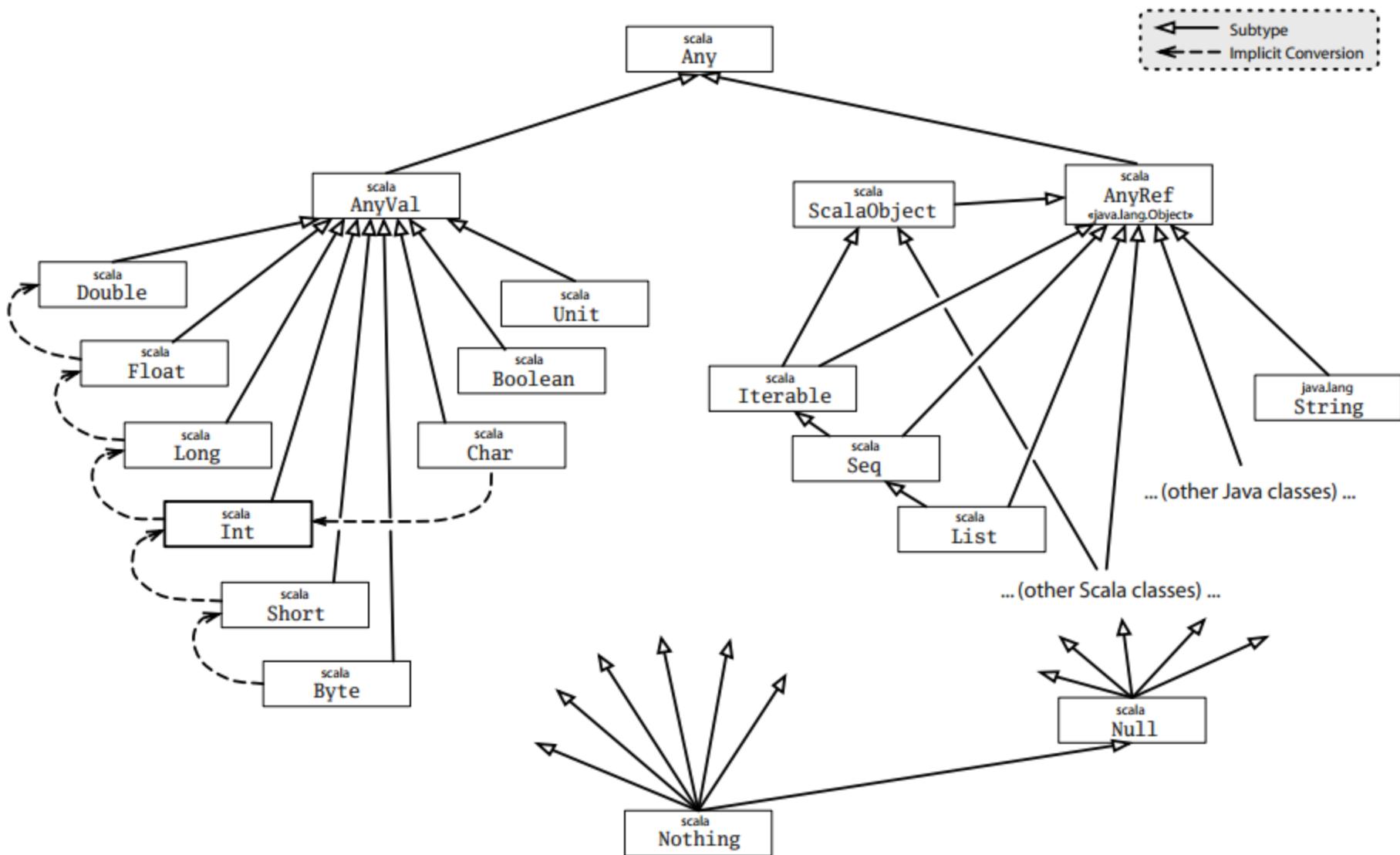
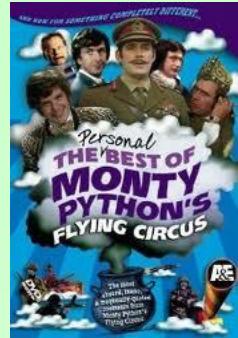


Figure 11.1 · Class hierarchy of Scala.

Python Language

- powerful dynamic programming language
- clear readable syntax (indentation and off-side rule)
- Strong introspection capability
- high-level dynamic types
- excellent “battery-included” libraries
- Why study Python?

fast, nimble
good for scripting



Python Example

Hello World:

```
print `Hello, World!'
```

Increment method:

```
def inc(x):  
    return x+1
```

Python Example

A List of Things:

```
lst = ['hello', 'there', 42, 0.2]

n = len(lst) # find length

if n<=0:
    print 'empty list'
elif x=1:
    print 'singleton'
else:
    print 'crowded list'
```

Repeating a list thrice:

```
lstlst = lst * 3
```

Prolog



- one of first language based on first-order logic
- it is used to define “relations” and relies on unification for execution
- Popular in AI and database applications (via datalog)
- Why study Prolog?

**relations, logic
and unification**

Prolog Example

Hello World:

```
main :- write('Hello, World!'),nl.
```

Increment method:

```
inc(X,Res) :- Res is X+1.
```

A Prolog Example

Facts (e.g. database):

```
parentOf(tom, sally).  
parentOf(tony, ale).  
parentOf(tony, alfred).
```

Derived Relations (e.g. query):

```
sibling(X,Y) :- parentOf(Z,X), parentOf(Z,Y).  
grandparent(X,Y) :- parentOf(X,Z), parentOf(Z,Y).  
grandfather(X,Y) :- male(X), grandparent(X,Y).
```

How to Pass CS2104

Expressible in Prolog:

```
pass2104(X) :-  
    attend_lecture(X,2104) ,  
    attend_tutorial(X,2104) ,  
    do_assignment(X,2104) ,  
    attempt_exam(X,2104) .
```

Untyped Lambda Calculus

- Extremely simple programming language which captures *core* aspects of computation and yet allows programs to be treated as mathematical objects.
- Focused on *functions* and applications.
- Invented by Alonzo (1936,1941), used in programming (Lisp – 2nd oldest language) by John McCarthy (1959).
- Why is it significant?

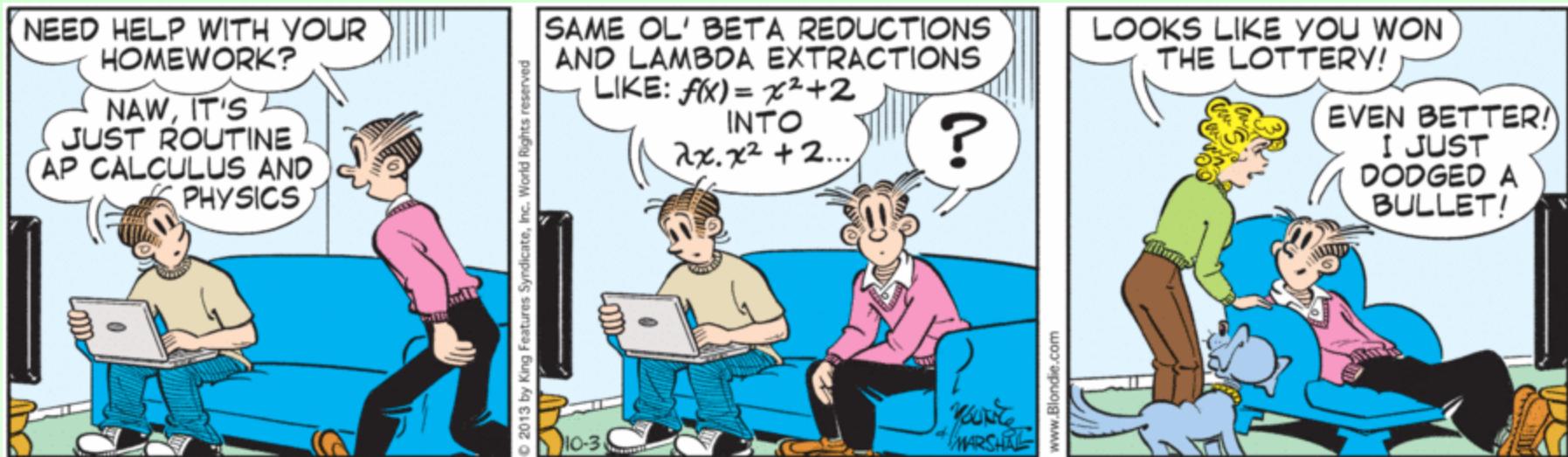


**Basis of
Computability**

Increment Method

With integer primitive, increment method can be written as:

$$fx = x+1 \quad \rightarrow (\lambda x . x+1)$$



Syntax

In purest form (no constraints, no built-in operations), the lambda calculus has the following syntax.

| | |
|-----------------|-------------|
| $t ::=$ | terms |
| x | variable |
| $\lambda x . t$ | abstraction |
| $t t$ | application |

This is **simplest** universal programming language!

Program vs Values

- Programs –expressions of lambda calculus
- Values – final irreducible expression.

Computation:



Examples of Lambda Expressions

- Function with one parameter:

$$\lambda x . x$$

- Function with two parameters:

$$\lambda x y . x$$

- Function that returns a function:

$$\lambda x . (\lambda y . x)$$

- Function application/call:

$$(\lambda x . x) y \rightarrow y$$

How Expressible is Lambda Calculus?

- Very expressive!
 - Boolean
 - Integer
 - Functions
 - Recursion
 - Data structures
 - Loops!
 - It is Turing-complete

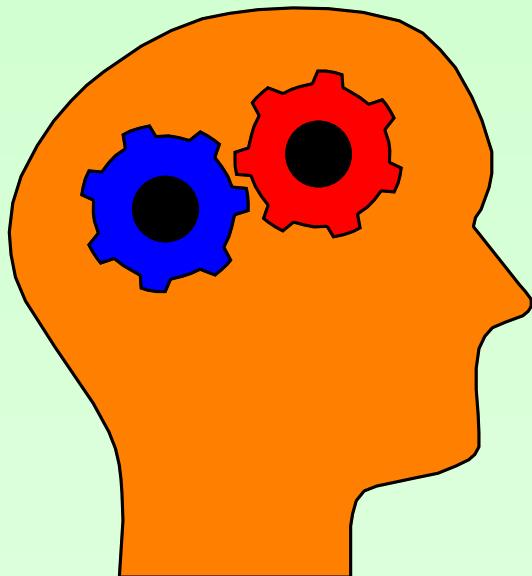
Non-terminating Loop

$$\begin{aligned} & (\lambda \text{ x. } \text{ x } \text{ x}) \, (\lambda \text{ x. } \text{ x } \text{ x}) \\ \rightarrow & (\lambda \text{ x. } \text{ x } \text{ x}) \, (\lambda \text{ x. } \text{ x } \text{ x}) \\ \rightarrow & (\lambda \text{ x. } \text{ x } \text{ x}) \, (\lambda \text{ x. } \text{ x } \text{ x}) \\ \rightarrow & \dots \\ \rightarrow & (\lambda \text{ x. } \text{ x } \text{ x}) \, (\lambda \text{ x. } \text{ x } \text{ x}) \\ \rightarrow & \dots \end{aligned}$$



CS2104: Programming Languages Concepts

Lecture 2 : Functions, Binders and Layout Rule



*“FP, Binders and Layout Mechanism
with Haskell”*

Lecturer : Chin Wei Ngan

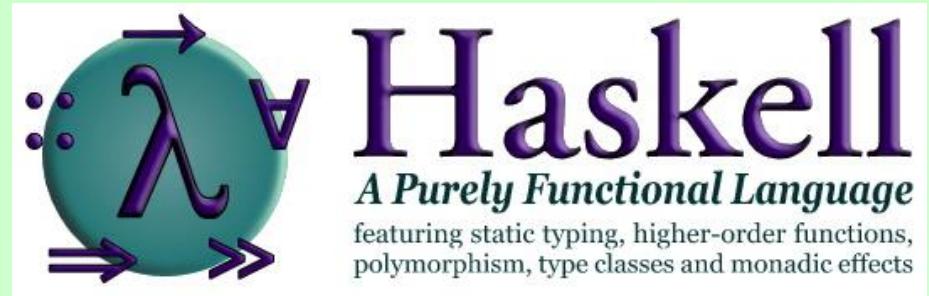
Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Topics

- Declarative Programming Concepts
- Writing Functions
- Local Bindings
- Typeful Programming

Advanced Language - Haskell



- Strongly-typed with polymorphism
- Higher-order functions
- Pure and Lazy Language.
- Algebraic data types + records
- Exceptions
- Type classes, Monads, Arrows, etc
- Advantages : concise, abstract, pure
- Why use Haskell ?

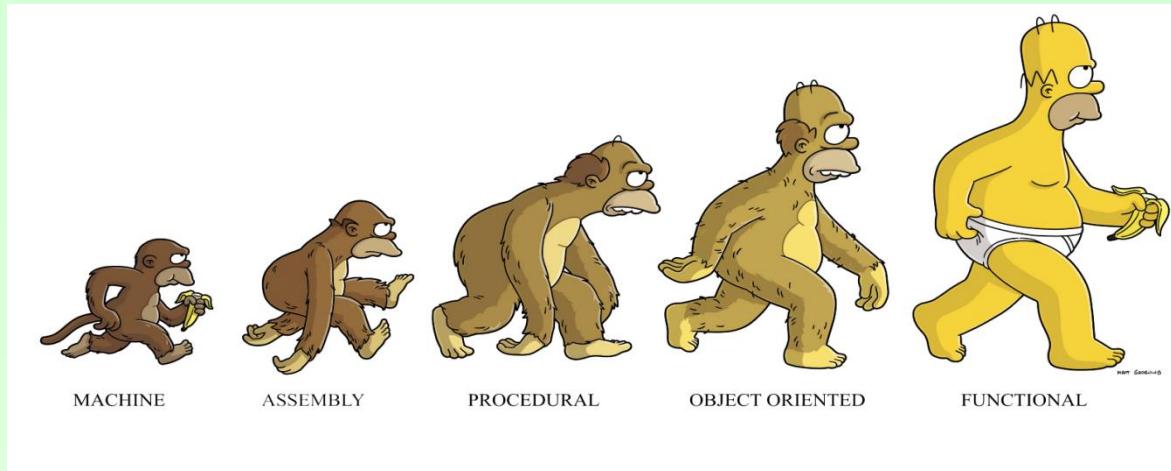
Cool, clean
and Pure

What is Purely Functional?

No updates and side-effects.



Easier debugging and better abstraction



Declarative Programming Model Philosophy

- Ideal of declarative programming
 - say **what** you want to compute
 - let computer find **how** to compute it
- More pragmatically
 - let the computer provide more support
 - free the programmer from some burden

Properties of Declarative Models

- Focus on functions (or relation) which compute when given some data structures as inputs
- Widely used
 - functional languages: LISP, Scheme, ML, Haskell, ...
 - logic languages: Prolog, Mercury, ...
 - representation languages: XML, XSL, ...
- Stateless programming
 - no update of data structures
 - Simple data transformer

Pure Functions

- Pure functions are *mathematical* functions whose outputs depend and solely on its inputs.
- Declarative Haskell
 - One of the purest language.
 - data structures is immutable and moreover is lazy by default.

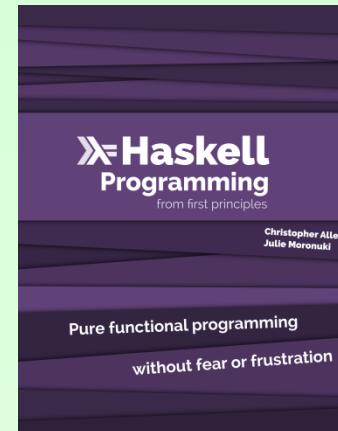
Reference

- Haskell home-page
www.haskell.org/documentation

- A fast Haskell tutorial

<https://www.schoolofhaskell.com/school/starting-with-haskell/haskell-fast-hard>

- A gentle and comprehensive Haskell book at haskellbook.com



Two Basic PL Concepts

- Writing Functions
- Local Bindings

Example - Haskell Program

- Computing Factorial.

-- written using conditional construct

```
fact :: Int -> Int
fact n = if n==0 then 1
         else n * (fact (n-1))
```

-- written using pattern-matching

```
fact1 :: Int -> Int
fact1 0 = 1
fact1 n = n * fact1 (n-1)
```

-- Int denotes bounded integer (with possible overflow)

```
*Main> fact 20
2432902008176640000
*Main> fact 30
-8764578968847253504
```

Example - Haskell Program

- Factorial with Infinite Precision

-- Integer denotes integer of arbitrary precision

```
fact2 :: Integer -> Integer  
fact2 0 = 1  
fact2 n = n * fact2(n-1)
```

-- No more overflows

```
*Main> fact2 20  
2432902008176640000  
*Main> fact2 30  
265252859812191058636308480000000  
*Main> fact2 40  
815915283247897734345611269596115894272000000000
```

Example - Haskell Program

- Mixed Precision

```
fact3 :: Int -> Integer  
fact3 0 = 1  
fact3 n = n * fact3(n-1)
```

-- However, we get a type error. Why?

```
lein.hs:16:11:  
  Couldn't match expected type `Integer' with actual type `Int'  
  In the first argument of `(*)', namely `n'  
  In the expression: n * fact3 (n - 1)  
  In an equation for `fact3': fact3 n = n * fact3 (n - 1)
```

Example - Haskell Program

- Fix problem using type conversion

```
-- fromIntegral :: Int -> Integer  
  
fact3 :: Int -> Integer  
fact3 0 = 1  
fact3 n = (fromIntegral n) * fact2(n-1)
```

-- Type in Haskell is actually much more general ...more details later.

```
*Main> :t fromIntegral  
fromIntegral :: (Integral a, Num b) => a -> b
```

-- For example, can also convert to floating point numbers ...

```
*Main> (fact 20)  
2432902008176640000  
*Main> (fromIntegral (fact 20))::Double  
2.43290200817664e18
```

Example - Haskell Program

- Finite and infinite lists.

a type variable

```
data List a = Nil | Cons a (List a)
```

```
finite_list :: List Int
```

```
finite_list = Cons 1 (Cons 2 (Cons 3 Nil))
```

finite list of three elements

```
infint :: Int -> List Int
```

```
infint n = Cons n (infint (n+1))
```

infinite list starting from [n,n+1,n+2,...]



Local Binding

- Binders are a special class of constructs which declare **new identifiers** and their possible values.
- Each binder has some lexical scope where the identifier is visible.
- Values of the binders may be either *immutable* or *mutable*.
- Examples of name binders
 - Local declarations
 - Method declaration (parameters)
 - Pattern-Matching

Local Binder in Haskell

- Local variable binders in Haskell are immutable

```
let x = 3::Integer  
in x*2
```

- where x is a local immutable variable denoting a value with a fixed scope.
- In general:

```
let v = e1  
in e2
```

For Haskell, scope of **v** is in both **e1** and **e2**.

Local Binder in Haskell

- Let binding in Haskell is recursive

```
let x = 1:x in  
in x
```

which is an infinite list of [1,1,1,1,...]

- Can write recursive function:

```
let f = \ n ->  
    if n==0 then 1  
    else n * (f (n-1))  
in f
```

- Be careful to avoid infinite loop:

```
let x = 1+x  
in x*2
```

Local Binder in OCaml

- Let binder in OCaml is non-recursive by default

```
let x = 1
  in let x = x+2
     in x
```

which will return 3.

- For OCaml:

```
let v = e1
  in e2
```

The scope of **v** is only in **e2**.

- Recursive let requires an extra keyword :

```
let rec v = e1
  in e2
```

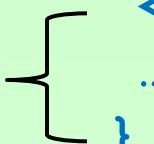
The scope of **v** is now in both **e1** and **e2**.

Local Binder in C

- Local binder in C is *mutable* and *non-recursive*

{ ...;
 <type> <id>;
 ...
}

scope of
local variable



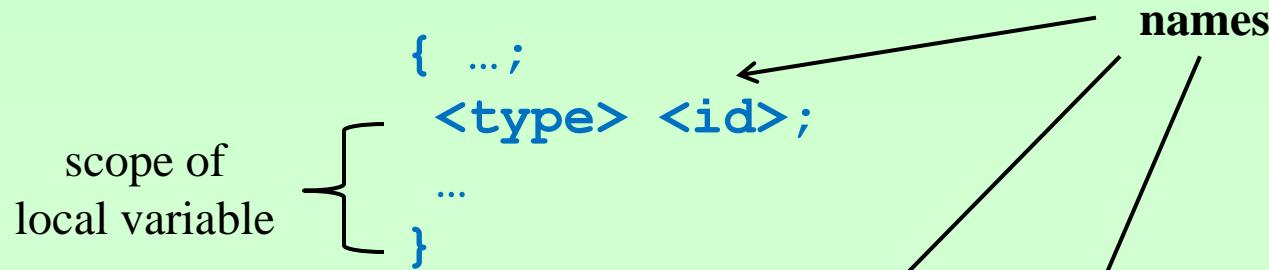
- Declare variable without a value. Assign variable to a new value with the assignment statement.

```
{ ...;
  int x;
  x = 1;
  x = x + 3
  ...
}
```

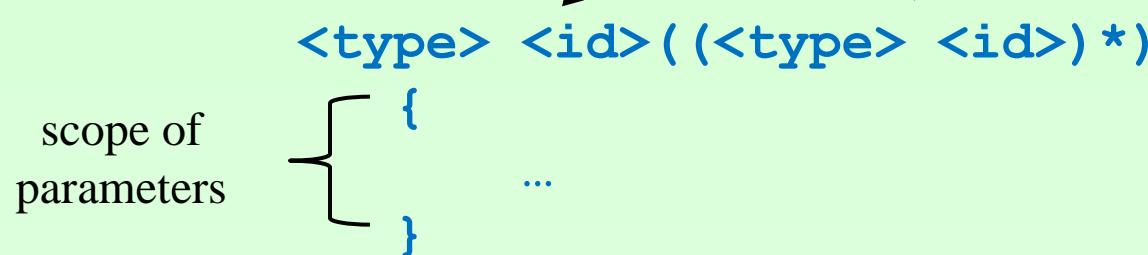
- Here, red **x** denotes previous value of **x**

Binders (in C)

- Block Declaration



- Method Declaration



Typeful Programming in Haskell

- Every expression has a type.
- We use ascription ($e :: t$) to force an expression e to have a given type t

```
let x = 3::Integer

let y = [1,2,3,4]::[Int]

let t = ("Tom", 2, 4.5)
-- type of t above will be inferred
-- as ([Char], Integer, Double)
```

Typing Comparison

- Typeful programming also in OCaml which has both object and module types, but lack type classes mechanism of Haskell.
- Both type inference and type checking possible.
- Weak typing in C with unsafe casting.
- Dynamic typing in Javascript and Python

Functions

- Functions are also values

```
let add x y = x + y
double x = (add x x)
quad x = (double x)+(double x)
```

- Layout rule supports multiple and recursive declarations
- Declarations may be nested:

```
let quad x =
    let add x y = x + y
        double x = add x x
    in (double x)+(double x)
```

Here, `add` and `double` are local functions, while `quad` is a global declaration.

Layout Rule

- Haskell uses two dimensional syntax to help reduce syntactic separators where declarations are “aligned”

```
let y      = a+b  
    f x    = (x+y)/y  
in f c + f d
```

is being parsed as:

```
let { y      = a+b  
      ; f x    = (x+y)/y }  
in f c + f d
```

- Rule : Next character after keywords `where`/`let`/`of`/`do` determines the starting columns for declarations.
Starting *after* this point continues a declaration, while starting *before* this point terminates a declaration.
- See <https://en.wikibooks.org/wiki/Haskell/Indentation>

Recursive Functions in Haskell

- Declarations are mutual recursive by default.

```
let fact n =
    if n=1 then 1
    else n * fact(n-1)
```

- Mutual-recursive functions can be defined and are aligned based on the layout rule

```
let foo n =
    if n<=1 then 1
    else foo(n-1) + goo(n-2)
goo n =
    if n<=1 then 1
    else goo(n-1) + foo(n-2)
```

alignment

- Note that **goo n-1** is parsed as **(goo n) -1**

Recursive Methods in OCaml

- Declarations are *non-recursive* by default. Add **rec** keyword to capture recursive declaration.

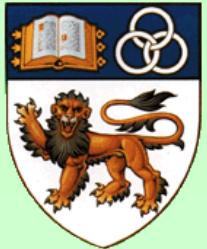
```
let rec fact n =
    if n=1 then 1
    else n * fact(n-1);;
```

- Mutual-recursive functions can be defined simultaneously with the help of the **and** keyword

```
let rec foo n =
    if n<=1 then 1
    else foo(n-1) + goo(n-2)
```

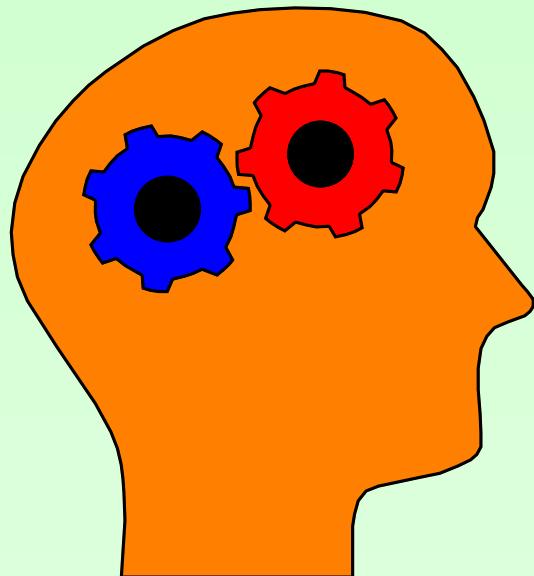
```
and goo n =
    if n<=1 then 1
    else goo(n-1) + foo(n-2);;
```

- Note that **goo n-1** is parsed as **(goo n) -1**



CS2104: Programming Languages Concepts

Lecture 3 : Data Types and Control Constructs



*“Types, Constructs and Recursion
and Type Classes”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Topics

- Data Types
- Efficient Recursion
- Type Classes



Program = Data + Algorithm

Primitive and User-Defined Types

Primitive Types

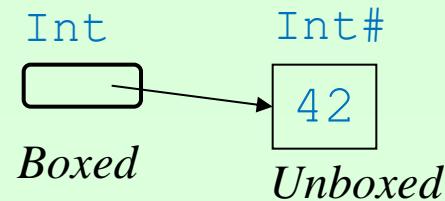
- Built-In Types of Languages
- Directly implemented in hardware (usually).
- Simpler types
 - e.g. boolean, integer, float, char, string, arrays
- Important for performance
- May include Standard Libraries (extensible)
- Language *without* primitive type
 - Lambda Calculus
 - Use just functions!

Primitive Types (C Language)

- Built-In Types
 - int (bounded integer, e.g. `int x = 42`)
 - float (single precision floating, e.g. `float r = 4.2`)
 - double (double precision floating point)
 - char (e.g. `char Letter; Letter = 'x';`)
 - arrays (e.g. `int arr[10];`)
- Modifiers
 - short, long (e.g. `short int, long double`)
 - signed, unsigned
- `void` can be considered a type with a single value
- `void*` denotes pointer with unspecified type
- boolean implemented as integer
(0 for false, non-0 for true)

Primitive Types (Haskell)

- Types are pretty high level in Haskell and are essentially boxed types
 - `data Bool = False | True`
-- e.g. `b = True`
 - `type String = [Char]`
 - `data Int = GHC.Types.I# GHC.Prim.Int#`
 - `data Float = GHC.Types.F# GHC.Prim.Float#`
 - `data Double = GHC.Types.D# GHC.Prim.Double#`
 - `data Char = GHC.Types.C# GHC.Prim.Char#`
- Unboxed types are built-in but seldom used, except when implementing the compiler using `-fglasgow-exts`



- Similar to objects versus primitives.

Boxed vs Unboxed Types

- Support Polymorphism
- Support Lazy Evaluation
- Programmer only sees boxed types
- Compiler writers or system programmer can optimize boxed type usages by converting them to the more efficient unboxed types.

User-Defined Types

- User can design their own types
- Support more complex data structures
- Examples:
 - Ordinal type
 - Record type
 - Pointer type
 - Tuple type
 - Union types
 - Algebraic Data type
 - Polymorphic/Generic type
 - Sum vs Product types

Ordinal Types

- An ordinal type is a finite *enumeration* of distinct values that can be associated with positive integers.
- An example in Ada is

```
type DAYS is (Mon,Tue,Wed,Thu,Fri,Sat,Sun)
```

```
if today>Fri then
    print "Hippie"
else
    print "Oh..."
```



- If values can occur in more than one ordinal type, it is referred to as *overloaded literals*, but may be difficult to determine which type such values belong to.

Ordinal Types in Haskell

- Enumeration is a special case of algebraic data type.

```
data DaysObj = Mon | Tue | Wed | Thu | Fri | Sat | Sun  
deriving (Show, Enum, Eq)
```

- Enum** class supports a range of useful methods:

```
class Enum a where  
    succ :: a -> a  
    pred :: a -> a  
    toEnum :: Int -> a  
    fromEnum :: a -> Int  
  
    ..
```

- Show** class supports printing:

```
class Show a where  
    show :: a -> [Char]
```

- Eq** class supports `==` and `/=`

Ordinal Types in Scala

- Scala enumeration can be done through sealed case objects.

```
object DaysObj {  
    sealed trait Days  
    case object Mon extends Days  
    case object Tue extends Days  
    case object Wed extends Days  
    case object Thu extends Days  
    case object Fri extends Days  
    case object Sat extends Days  
    case object Sun extends Days  
    val daysOfWeek = Seq(Mon, Tue, Wed, Thu, Fri, Sat, Sun)  
}
```

- Sealed trait can be extended only in a single file.
- Compiler can thus emit warning if a match is not exhaustive

Ordinal Types

- Non-exhaustive pattern. (e.g. `Tue` missing)

```
let weekend x = case x of {  
    Mon -> False;  
    Sat -> True;  
    Sun -> True;  
}
```

Leads to accidental exception, Use `ghci -Wall`

Compiler option in code:

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
```

- Exhaustive Pattern.

```
let weekend x = case x of {  
    Sat -> True;  
    Sun -> True;  
    _ -> False;  
}
```

Each pattern contain only one case. No multi-case in Haskell

Pattern Matching : Haskell vs OCaml

- Haskell allows guards but not OCaml

```
weekend x
| x==Sat || x==Sun -> True
| otherwise   -> False
```

- OCaml allows multiple cases but not Haskell

```
let weekend x = match x of {
  Mon -> False;
  Sat | Sun -> True;
}
```

- No language is perfect !
We just need to learn to live with it.

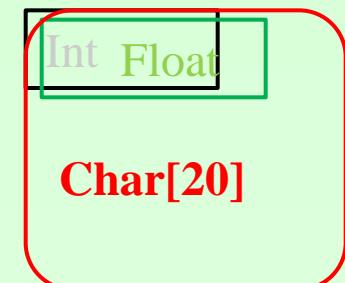
Union Types (in C)

- Union type is a type that may store different type of values within the same memory space but at different times/instances.
- We can define a union of many members but only one member can exists at any one time. Format:

```
union [union_tag] {  
    (type id)+  
} id+ ;
```

- An example where the fields are overlaid.

```
union Data {  
    int i;  
    float f;  
    char str[20]  
};
```



Algebraic Data Type (in Haskell)

- A more systematic way to capture union types safely is to use algebraic data types which uses data constructor tags.

```
data Data = I Int | F Float  
          | S String deriving Show
```

- With this, we can build objects of different values.

```
let v1 = I 3;;  
let v2 = F 4.5;;  
let v3 = S "CS2104";;
```

Algebraic Data Type (in Haskell)

- Generic printer.

```
let print_data v =
    case v of
        I a -> show a
        F v -> show v
        S v -> v
```

- Note pattern-matching is able to obtain type-safe values.
- What is the type of this function?
- What is the difference between **print_data** and **show**?

Tuple Types (Haskell)

- A special case of algebraic data type is the *tuple* type. This is polymorphic and has only a single data constructor. An example is the triple:

```
data (,,) a b c = (,,) a b c  
let stud1 = ("John","A1234567J",2013);;
```

- We can write *type synonym*, as follow:

```
type Student = (String, String, Int)  
type Pair a b = (a, b)  
type String = [Char]
```

Tuple Types (Haskell)

- Like algebraic data type, the component of tuples can be accessed using pattern-matching:

```
let (name,_ ,yr) = stud1;  
  
case stud1 of  
 (name,_ ,yr) -> ... ; ;
```

Type Synonym (Haskell)

- Type synonyms cannot be recursive.

```
type Node = (Int, Node);;
```

Error: Cycle in type synonym declarations

- This is so as it leads to infinite expansions.

Record Types

- Record is a collection of values of possibly different types. In C, this is implemented as structures

```
struct [structure_tag] {  
    (type id)+  
} id+ ;
```

- An example:

```
struct Student {  
    char name[20];  
    char matric[10];  
    int year;  
} s1, s2 ;
```

Record Types (C)

- Can access the fields by the dot notation

Examples : `s1.year or s2.matric`

Access is efficiently done via offset calculation.

- Operations on records, e.g. assignment

`s1 = s2;`

Entire record is copied or passed as parameters.

Record Types (Haskell)

- Record type can also be supported in Haskell as an extension to algebraic data type

```
data Student = Student {  
    name :: String;  
    matrix :: String;  
    year :: Int  
} ;;
```

- Automatically derives name, matrix and year as access methods.

```
name :: Student -> String  
matrix :: Student -> String  
year :: Student -> Int
```

Record Types (Haskell)

- Pattern-matching with records.

```
data X = A | B {name :: String}
        | C {x::Int, y::Int, name::String}
```

- Automatically derives `name`, `x` and `y` as access methods
- Can use pattern-matching in record:

```
myfn :: X -> Int
myfn A = 50
myfn B{} = 200
myfn C{x=v} = v
```

Record Types (OCaml)

- Record type is a separate type in OCaml

```
type student = {  
    name : string;  
    matrix : string;  
    year : int  
} ;;
```

- Fields are immutable, by default. Mutable fields are also allowed.
- Use pattern-matching to access *selected* components

```
match s with  
| {name = n; year = y} -> ...
```

- What are types of **n** and **y**?

Pointer Type (in C)

- A pointer type is meant to capture the address of an object value. There is also a special value `nil` that is not a valid address.
- An example in C with pointer types are:

```
int count, init;  
int *ptr;  
...  
ptr = &init;  
count = *ptr;
```

| | |
|-------|---|
| count | 4 |
| init | 6 |
| ptr | |

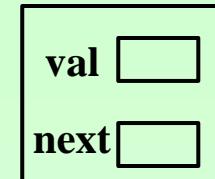
- The `*` operation will dereference a pointer type.

Pointer Type (in C)

- Pointer types are crucial for recursive data type.
- An example in C is:

```
struct node {  
    int val;  
    struct node *next;  
} list;
```

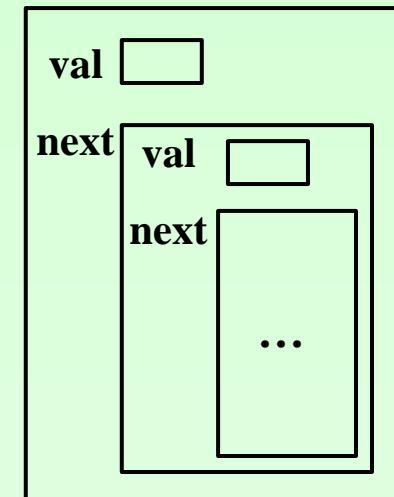
list



- We cannot use below. Why?

```
struct node {  
    int val;  
    struct node next;  
} inf_list;
```

inf_list

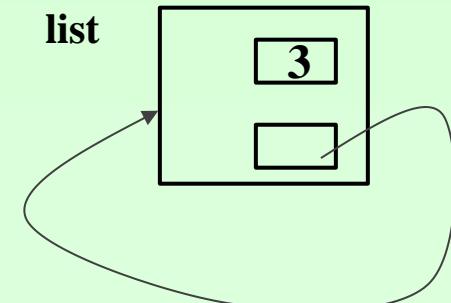


Pointer Types (in Haskell?)

- No need for pointer types in Haskell.
- By default, each algebraic data type is already implemented as a pointer to a boxed value.
- Thus, recursive type is possible but can be infinite.

```
data Rnode = Rnode(Int, Rnode)
```

```
let list = Rnode(3, list)
```



Pointer Types (Haskell)

- We may use a general Maybe type

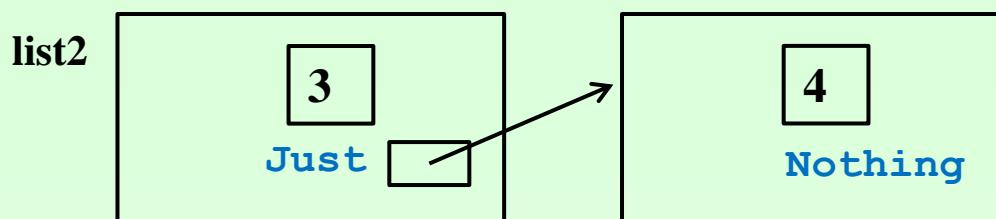
```
data Maybe a = Nothing | Just a
```

- Possibly finite data structure:

```
data RNode2 = RNode2 Int (Maybe RNode2)  
            deriving Show
```

- An Example

```
let list2 = RNode2 3 (Just (RNode2 4 Nothing))
```



Product vs Sum Types

- Fundamentally, there are two kinds of types.
- *Product* types that include tuples and records.
- *Sum* types that include ordinals and general algebraic data types.
- Product type is similar to conjunction $a \wedge b$

```
type Pair a b = (a, b)
```

- Sum type is similar to disjunction $a \vee b$.

```
type Either a b = Left a | Right b
```

Untyped Languages

- Untyped language essentially sums different types together, without using tags, e.g:

```
data Univ = Int | Float | ... | Array ...
```

- However, cannot distinguish between the sum type and its constituent types!

```
data Sum a b = a | b
```

```
let v1 = [3, "cs2104"];;
```

- Without tags, there is *no type safety* that can be ensured at both compile and run-time.

Type Classes

Type Classes and Overloading

- Parametric polymorphism works for all types.
However, *ad-hoc polymorphism* works for a *class of types* and can systematically support overloading.
- For example, what is the type for `(+)` operator?

```
(+) :: a -> a -> a  
(+) :: Int -> Int -> Int  
(+) :: Float -> Float -> Float
```

- What about methods built from `(+)` ?

```
double x = x+x
```

- Solution is to use a **type class** for numerics!

```
(+) :: Num a => a -> a -> a  
double :: Num a => a -> a
```

Equality Class

- Similar issue with equality. Though it looks like:

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

- Equality can be tested for any data structure but not functions!
- Solution : define a **type class** that supports the equality method, as follows:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y      = not (x==y) ← default definition
```

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

Members of Eq Type Class

- The class is open and can be extended, as follows:

```
instance Eq Integer where
  x == y    = x `integerEq` y
```

```
instance Eq Float where
  x == y    = x `FloatEq` y
```

- Recursive type can also be handled but elements may need to be given *type qualifiers*.

```
instance (Eq a) => Eq (Tree a) where
  Leaf a == Leaf b = a==b
  (Branch l1 r1) == (Branch l2 r2)
    = (l1==l2) && (r1==r2)
  _ == _           = False
```

More Members of Eq Type Class

- Similarly, we may use structural equality for List:

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

- One use of Eq class is:

```
elem :: (Eq a) => a -> [a] -> Bool
x `elem` []        = False
x `elem` (y:ys)    = (x==y) || (x `elem` ys)
```

- Exercise : define Set as an instance of the Eq class.

Class Extension

- Haskell supports the notion of extending from some base type class.

```
class (Eq a) => Ord a where
    (<) , (≤) , (≥) , (>)  :: a -> a -> Bool
    max, min                :: a -> a -> a
```

- This help reuse previous definitions from base class.
Default definitions are:

| | |
|------------|---|
| $x < y$ | $= x \leq y \ \&\& \ x \neq y$ |
| $x \geq y$ | $= y \leq x$ |
| $x > y$ | $= y < x$ |
| $\min x y$ | $= \text{if } x \leq y \text{ then } x \text{ else } y$ |
| $\max x y$ | $= \text{if } x \geq y \text{ then } x \text{ else } y$ |

Class Extension

- An example that use **Ord** class :.

```
quicksort :: (Ord a) => [a] -> [a]
```

- Actual **Ord** class in the prelude is based on:

```
data Ordering = EQ | LT | GT  
compare :: Ord a => a -> a -> Ordering
```

- Multiple inheritances supported which would allow more operations to be inherited from base classes.

```
class (Eq a, Show a) => C a where ...
```

Enumeration Class

- Class **Enum** supports sugar for arithmetic sequences:

```
class (Enum a)
  enumFromThen :: a -> a -> [a]
  fromEnum :: a -> Int
  toEnum :: Int -> a
```

- An example of arithmetic sequence :

```
[Red..Violet] ) [Red,Green,Blue,Indigo,Violet]
```

Show Class

- The **Show** class are for types that can be converted to character string.

```
class Show where
    show :: a -> String
    shows :: a -> String -> String
    show x = shows x ""
```

- An example of its use:

```
showTree      :: (Show a) => Tree a -> showTree
showTree (Leaf x)      = show x
showTree (Branch l r) = "<" ++ showTree l ++
                      "| " ++ showTree r ++ ">"
```

- Possible problem : *quadratic complexity*.

Show Class

- More efficient to use the accumulating method :

```
shows :: Show a => a -> String -> String
```

```
showsTree :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) z = shows x z
showsTree (Branch l r) z = `<` : showsTree l
                           (`|` : showsTree r (`>` : z))
```

- Can avoid use of accumulating parameter in the syntax via higher-order functions.

```
type ShowS = String -> String
showsTree :: (Show a) => Tree a -> ShowS
showsTree (Leaf x) = shows x
showsTree (Branch l r) = (`<` : ) . (showsTree l)
                           . (`|` : ) . (showsTree r) . (`>` :)
```

Read Class

- The converse problem of parsing data structure is handled by the `Read` class.

```
class Read a
  reads :: String -> [(a, String)]
```

- Empty answer denotes *failure* of parsing, while multiple answers denote *non-determinism*

```
type Reads a = String -> [(a, String)]
readsTree :: (Read a) => Reads (Tree a)
readsTree (`<` : s) = [(Branch l r, u)
  | (l, `|` : t) <- readsTree s, (r, `>` : u) <- readsTree t]
readsTree s          = [(Leaf x, t) | (x, t) <- reads s]
```

Read Class

- Problems (i) no white space (ii) punctuation symbols?
- Lexical analyser is provided in Prelude.

`lex :: ReadS String`

- Using lexer, we can build more robust parser:

```
readsTree s  = [ (Branch l r, x) | ("<",t) <- lex s,
                  (l, u)    <- readsTree t,
                  ("|", v)  <- lex u,
                  (r, w)    <- readsTree v,
                  (">", x)  <- lex w ]
++ [ (Leaf x, t) | (x,t) <- reads s]
```

Derived Instances

- Writing instances for standard type classes can be tedious, so Haskell allows automatic derivation.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
    deriving (Eq, Ord)
```

- This automatically derives the following lexicographic ordering for `Ord` class.

```
instance (Ord a) => Ord (Tree a) where
    (Leaf _) <= (Branch _) = True
    (Leaf x) <= (Leaf y) = x<=y
    (Branch _) <= (Leaf _) = False
    (Branch l r) <= (Branch l' r')
        = l<l' || (l==l' && r<=r')
```

Direct Handling of Errors (in Haskell)

Use Maybe type.

```
myDiv2 :: Float -> Float -> Maybe Float  
myDiv2 x 0 = Nothing  
myDiv2 x y = Just (x / y)
```

Use Either type (opposite of Pair a b)

```
myDiv3 :: Float -> Float -> Either String Float  
myDiv3 x 0 = Left "Division by zero"  
myDiv3 x y = Right (x / y)
```

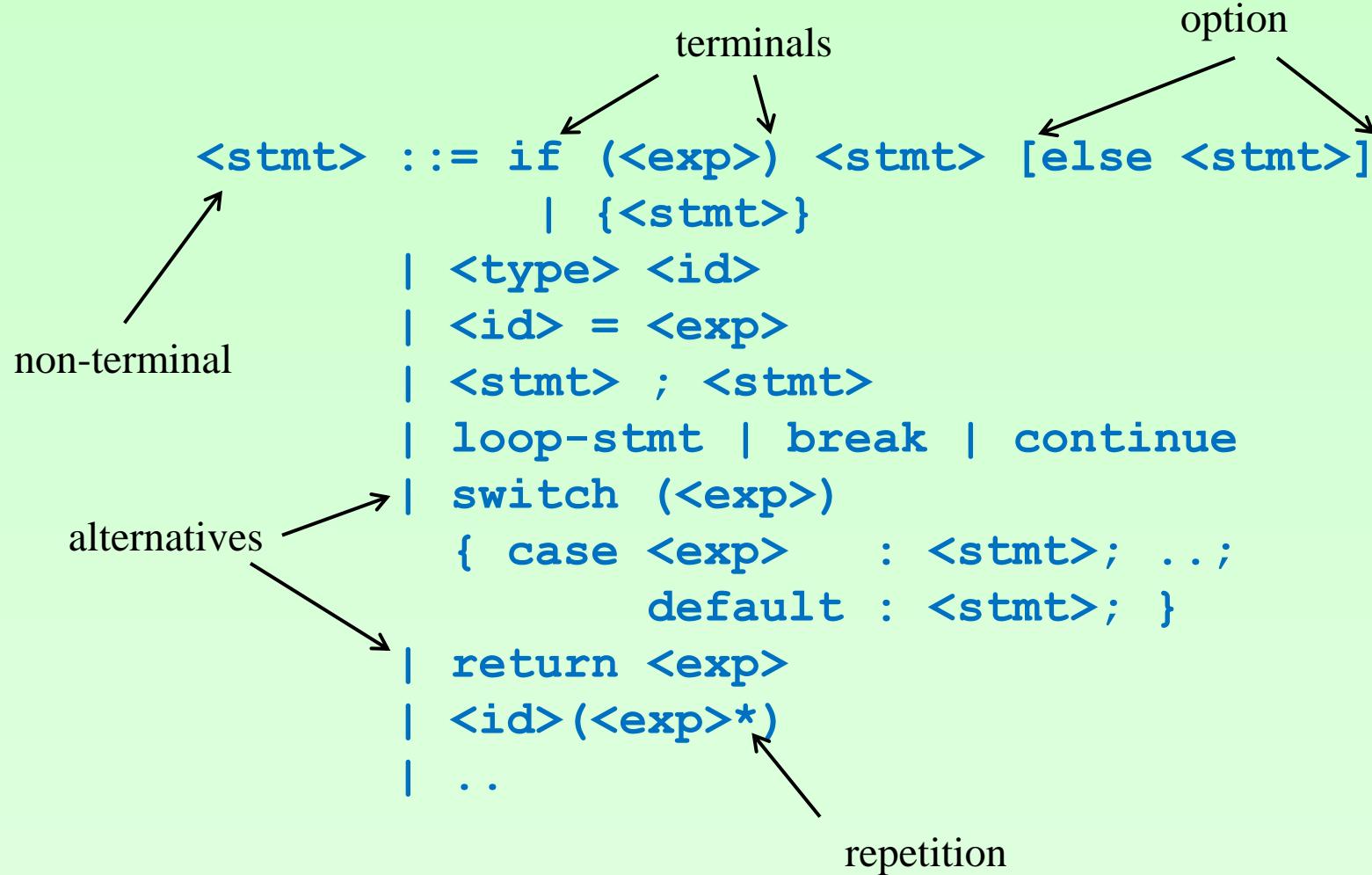
Language Constructs and Recursion

Language Constructs

- Language constructs are features used to build programs
- Statements vs Expressions
 - Statement executed for their *effects*
 - Expression computes a result and may have effects too.
 - Statement – a special case of expression
- Examples of Constructs
 - Blocks, Conditional, Sequences,
 - Exceptions, Loops, Calls

Language Construct (for C)

- Statement



Language Construct (for C)

- Loop Statement

```
<loop-stmt> ::=  
    for(<var_init>; <exp>; <var_upd>) <stmt>  
    | while (<exp>) <stmt>  
    | do {<stmt>} while (<exp>)
```

- Expression

```
<exp> ::= <id>  
    | <id>(<exp>*)  
    | <exp> ? <exp> : <exp>  
    | <op> <exp> | <exp> <op>  
    | <exp> <op> <exp>
```

Language Construct (for Haskell)

- Expression-Oriented Constructs

```
<exp> ::= if <exp> then <exp> else <exp>
| <id>
| <exp>:<type>
| <exp> <exp>
| (<exp>, . . . , <exp>)
| raise <exp>
| let <id>+ = <exp> in <exp>
| <exp> where (<id> = <exp>)+      ←
|   -- valid at top level defn
| case <exp> of {(<pat> -> <exp>)+} ←
| ...                                     one or more
```

- `let` declaration and `case` pattern contains binders that are immutable (bound values cannot be changed).

Functions

- Types for Functions
- Tupled vs Curried Functions
- Recursion and Tail-Recursion
- Naïve vs Efficient Functions

Types for Functions

- Every value has a type, including functions
- Some examples:

`(\ x -> x * x)` has type `Num a => a -> a`

`sum` has type `Num a => [a] -> a`

`length` has type `[a] -> Int`

`fst` has type `(a,b) -> a`

- Why didn't `length` method use `Integer`?
Why did `sum` method use the `Num` class?

Tupled vs Curried Functions

- Curried function takes one argument at a time, by returning a function that takes the next argument.

```
{- addA::Int->Int->Int -}  
let addA x y = x + y
```

- Tupled function takes all its arguments as a tuple/product.

```
{- addB::(Int,int)->Int -}  
let addB (x,y) = x + y
```

- Curried and tupled function are *isomorphic*.

```
addB ≡ \ (x,y) -> addA x y  
addA ≡ \ x -> \ y -> addB(x,y)
```

Partially Applied Functions

- Partially applied functions are those taking only some of their parameters.
- They result in functions that would take remaining arguments.

```
-- inc1 : Integer -> Integer  
let inc1 = addA 1
```

- To implement partial application for tupled method, you would need lambda abstraction to mimic it.

```
-- inc2 : Integer -> Integer  
let inc2 = \ y -> addB (2,y)
```

Topics

- Data Types
- Binders
- Efficient Recursion
- Type Classes

List Append

- `(++)` joins two lists together.
- Its has type `[a] -> [a] -> [a]`

```
xs ++ ys =  
  case xs of  
    [] -> ys  
    x:xs -> x:(xs ++ ys)
```

- Guess its time-complexity?

List Reverse

- `reverse` would reverse the content of its given list, so that the last element become first and vice-versa
- An easy implementation is:

```
reverse xs =  
  case xs of  
    [] -> []  
    x:xs -> (reverse xs) ++ [x]
```

- Is this efficient? Guess its time-complexity?

List Reverse (efficient and tail-recursive)

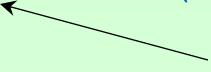
- A linear-time version that is also tail-recursive is shown below:

```
revit xs =  
  let aux xs acc =  
      case xs of  
        [] -> acc  
        x:xs -> aux xs (x:acc)  
  in aux xs []
```

- It uses an inner recursive function which kept a list of reversed elements so far in the `acc` parameter.
- How is its time-complexity? linear-time?

Tail-Recursion

- A function is tail-recursive if the recursive call is always the last operation in the recursive invocations.

```
let aux xs acc =  
    case xs of  
        [] -> acc  
        x::xs -> aux xs (x::acc) in ...  
  
last call in the recursive branch
```

- Tail-recursion is equivalent to Loop.
- Last recursive call can be compiled into a jump to the beginning of recursive method.

Inefficiency of Naïve Fibonacci

- An often-cited example of inefficiency of recursion is the fibonacci function.

```
let fib n =  
    if n<=1 then 1  
    else (fib(n-1))+(fib(n-2))
```

- However, this is an algorithmic problem (with many repeated calls) rather than the problem of recursion per se.

Tupled Fibonacci

- We can easily write an efficient version of fibonacci by using a recursive helper that returns two fibonacci values:

(fib n, fib(n-1))

- The efficient fibonacci is now implemented as:

```
let fib2 n =  
    let aux n =  
        if n<=0 then (1, 0)  
        else let (a,b)=aux(n-1)  
              in (a+b,a)  
        in fst(aux n)
```

- This method eliminates the redundant calls and can thus computes in linear-time. However, it is not *tail-recursive*.

Tail-Recursive Fibonacci

- We can easily implement a tail-recursive fibonacci function by adding two extra parameters to the recursive method, as follows:

```
let fib3 n =
    let rec aux n a b =
        if n<=0 then a
        else aux (n-1) (a+b) a
    in aux n 1 0
```

- This is equivalent to most loop-based implementation of fibonacci.
- Lesson : You can write efficient *recursive* functions. How about a logarithmic-time fibonacci?

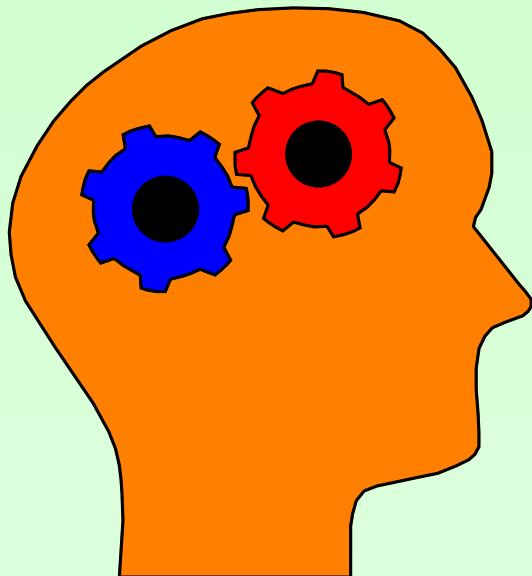
Topics

- Data Types
- Binders
- Efficient Recursion
- Type Classes



CS2104: Programming Languages Concepts

Lecture 4 : Higher-Order Functions



*“Programming with
First-Class Functions”*

Lecturer : [Chin Wei Ngan](#)
Email : chinwn@comp.nus.edu.sg
Office : COM2 4-32

Topics

- First-Class Functions
- Higher-Order Functions
 - Genericity/Parameterization
 - Fold (left and right) & Foldable
 - Map & Functor/Applicative
 - Composition
 - Pipeline
 - Application

Essence of FP

What is a Functional Language

A language
where functions
are first-class
citizens



Higher-Order Functions

- Like data structures, functions should be first-class:
 - It has a value and type
 - It can be passed as arguments
 - It can be returned as function result.
 - It can be constructed at run-time
 - It can be stored inside data structures
- Why are higher-order functions useful?
 - Can support code-reuse
 - Can support laziness
 - Can support data abstraction (see OO later)
 - Can support design patterns

Functions that Returns Functions

- Function results:

```
let add x = \ y -> x+y
```

- Equivalent to curried function:

```
let add x y = x+y
```

- Can also return different functions :

```
let add_mag x =
  if x>=0 then \ y -> x + y
  else \ y -> -x + y
```

Curried vs Uncurried Functions

- Type of Curried Function:

a → b → c

- Type of Uncurried (or Tupled) Function:

(a, b) → c

- These two functions are isomorphic and are interconvertible using:

```
curry :: ((a,b)->c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b)->c)
```

Functions that Returns Functions

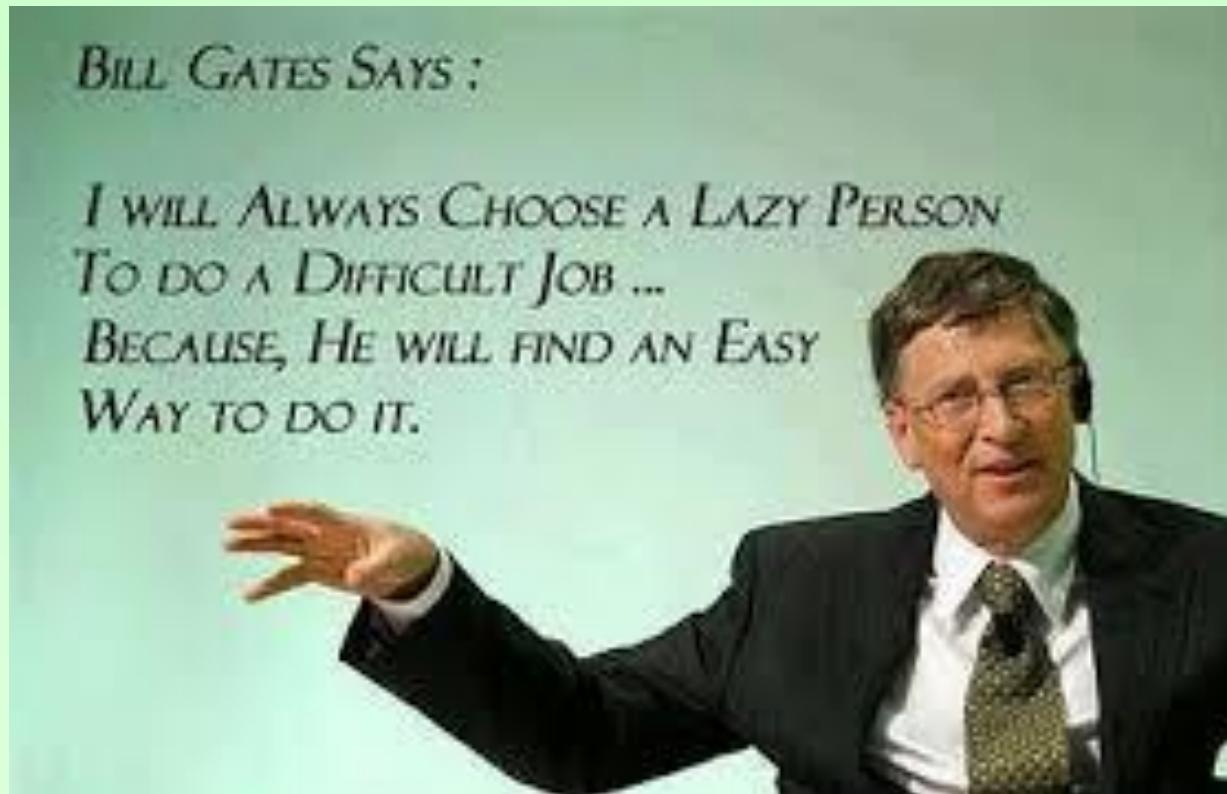
- Examples of Usages:

```
let add x = \ y -> x+y  
let inc = add 1  
let inc10 = add 10  
let dec = add (-1)  
let two = inc (1::Int)
```

- An Evaluation:

```
two  
→ inc 1  
→ (add 1) 1  
→ (\ y -> 1+y ) 1  
→ 1+1  
→ 2
```

Is Laziness Good?



Lazy Evaluation via Functions

- Any given expression e can be abstracted into a function $(\lambda () \rightarrow e)$.
- This is called a *closure* which provides a way to define an expression without evaluating it.
- To evaluate the expression, we simply apply it to $()$, as follows:

$$(\lambda () \rightarrow e) () \implies e$$

Evaluation of expression is delayed to application.
Update of *closure* is supported to reuse result of evaluation.
If function is not applied, the expression is not evaluated.

Lazy Evaluation

- This is the default evaluation for Haskell.
- It applies to both function calls and also let construct.
- Lazy evaluation allows us to handle, as long as non-terminating **bot** computation is not evaluated (or required) by its function

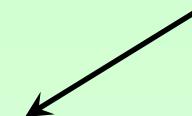
```
let bot = bot in ....  
f(...,bot,...)
```

Infinite Data Structures

- Circular structures are more space efficient :

`ones = 1 : ones`

list
comprehension



`fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]`

`zip (x:xs) (y:ys) = (x,y) : (zip xs ys)`
`zip xs ys = []`

- This circular fibonacci function can be computed very efficiently.

Strict Data Constructors

- Strict evaluation is the converse of lazy evaluation.
- In Haskell, if strict evaluation is needed, we can mark it with `!`. This can save on memory for building closures and thus minimise memory leaks.
- An example:

```
data RealFloat a => Complex a = !a :+ !a
```

- With this, `1 :+ bot` is then equivalent to just `bot`

Strict Evaluation

- To evaluate e_1 strictly, Haskell allows:
 - $e_1 \text{ `seq' } e_2$
 - $\text{case } e_1 \text{ of } ...$
- GHC extension also allows :

$f \$! e$

where e is strictly evaluated

$\text{let } f \; x \; !y \; !z = ...$

where y, z are strictly evaluated

Genericity/Parameterization

- To make a function generic is to let any specific entity (i.e. operation or value) in the function body become an argument
- This parameter abstraction can help us obtain more generic program code.

Two Similar Examples

- Function to sum a list of numbers.

```
let sum xs =  
    case xs of  
        [] -> 0  
        y:ys -> y+(sum ys)
```

- Function to multiply a list of numbers.

```
let prod xs =  
    case xs of  
        [] -> 1  
        y:ys -> y*(prod ys)
```

- Examples :

sum [1,2,3,4] → 1+2+3+4+0

prod [1,2,3,4] → 1*2*3*4*1

Genericity

- Replace each constant (that differs) by a parameter.
- Replace each function (that differs) by a parameter.

```
let sum xs =  
  case xs of  
    [] -> 0  
    y:ys -> y + (sum ys)
```

- Generalize to :

```
let foldr xs =  
  case xs of  
    [] -> z  
    y:ys -> f y (foldr ys)
```

Generic Fold Method

- Generalization of sum and prod gives fold.

```
let foldr f z xs =
  let aux xs =
    case xs of
      [] -> z
      y:ys -> f y (aux ys)
  in aux xs
```

- Usages :

```
let sum xs = foldr (+) 0 xs
```

```
let prod xs = foldr (*) 1 xs
```

Higher-Order Types

- Type of sum/prod.

```
let sum xs = ...
```

```
Type of sum :: Nat a => [a] -> a
```

```
let rec prod xs = ...
```

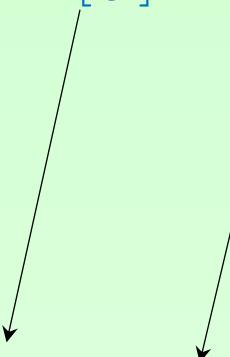
```
Type of prod :: Nat a => [a] -> a
```

- Type of fold :

```
let foldr f z xs = ...
```

```
Type of foldr ::
```

```
(a -> z -> z) -> z -> [a] -> z
```



Example Execution

- Example of sum:

```
sum [1,2,3]
→ foldr (+) 0 [1,2,3]
→ aux [1,2,3]
→ + 1 (aux [2,3])
→ + 1 (+ 2 (aux [3]))
→ + 1 (+ 2 (+ 3 aux []))
→ + 1 (+ 2 (+ 3 0))
```

```
foldr f z xs =
let aux xs =
  case xs of
    | [] -> z
    | y:ys -> f y (aux ys)
  in aux xs
```

- Example of prod:

```
prod [1,2,3]
→ foldr (*) 1 [1,2,3]
→ aux [1,2,3]
→ * 1 (aux [2,3])
→ * 1 (* 2 (aux [3]))
→ * 1 (* 2 (* 3 aux []))
→ * 1 (* 2 (* 3 1))
```

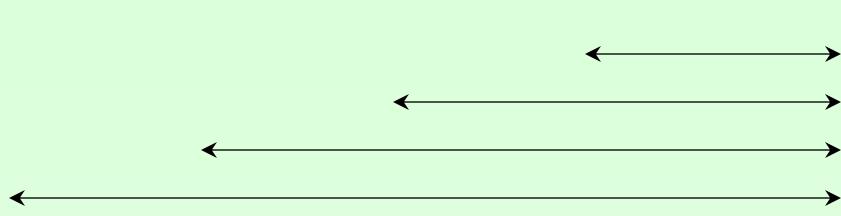
Right Fold Method

```
foldr f z xs =  
    let aux xs =  
        case xs of  
            [] -> z  
            y:ys -> f y (aux ys)  
    in aux xs
```

- Actually foldr denotes fold_right:

```
foldr f z [x1,x2,x3,x4]
```

→ $f\ x_1\ (f\ x_2\ (f\ x_3\ (f\ x_4\ z)))$

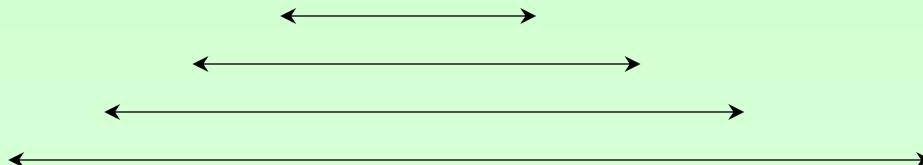


Left Fold Method

- An example of foldl method which folds leftwards:

```
foldl f z [x1, x2, x3, x4]
```

```
→ f (f (f (f z x1) x2) x3) x4
```



left is here!

Fold Left Method

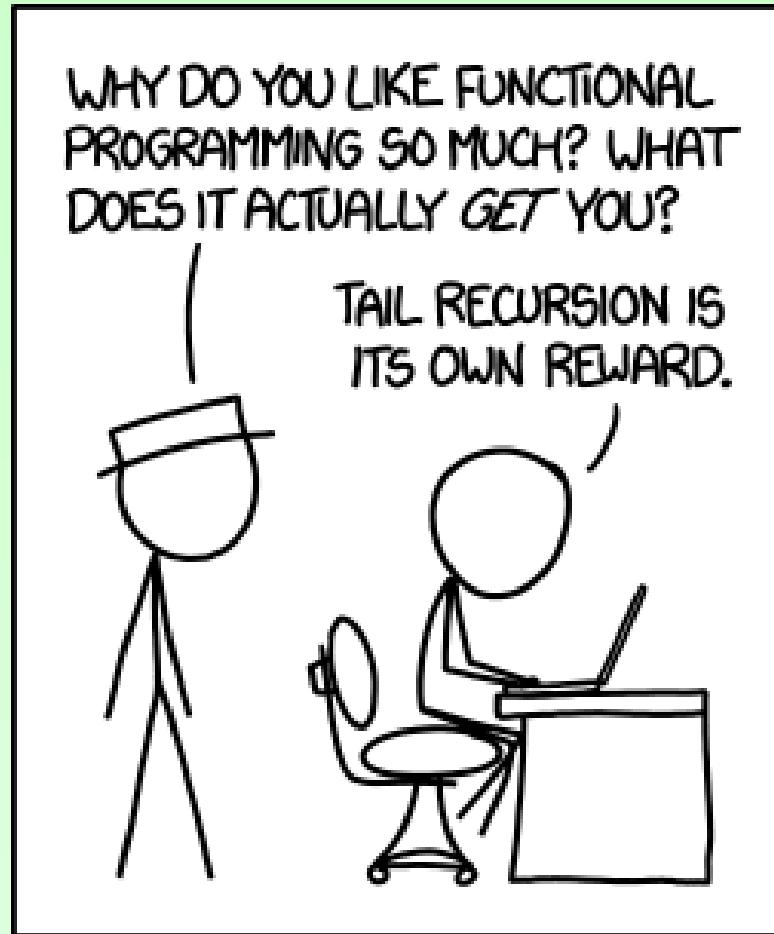
- Implementation of `foldl` method:

```
let foldl f z xs =
    let aux acc xs =
        case xs with
        | [] -> acc
        | y:ys -> aux (f acc y) ys
    in aux z xs
```

Key property : *tail-recursive!*

Type of `foldl` ::
 $(z \rightarrow a \rightarrow z) \rightarrow z \rightarrow [a] \rightarrow z$

Fold Left is Tail-Recursive



Example Execution

- Using *right fold*:

```
let foldr f z xs =  
  let rec aux xs =  
    match xs with  
    | [] -> z  
    | y::ys -> f y (aux ys)  
in aux xs
```

```
sum [1,2,3]  
→ foldr (+) 0 [1,2,3]  
→ aux [1,2,3]  
→ + 1 (aux [2,3])  
→ + 1 (+ 2 (aux [3]))  
→ + 1 (+ 2 (+ 3 aux []))  
→ + 1 (+ 2 (+ 3 0))  
→ 6
```

- Using *left fold*:

```
let foldl f z xs =  
  let rec aux acc xs =  
    match xs with  
    | [] -> acc  
    | y::ys -> aux (f y acc) ys  
in aux z xs
```

```
sum [1,2,3]  
→ foldl (+) 0 [1,2,3]  
→ aux 0 [1,2,3]  
→ aux (0+1) [2,3]  
→ aux 1 [2,3]  
→ aux 3 [3]  
→ aux 6 []  
→ 6
```

Fold Left or Fold Right?

- Can be transformed to each other when the reduction f operator is *associative* :

$$f\ a\ (f\ b\ c) = f\ (f\ a\ b)\ c$$

- Typically, z is the unit of f:

$$\begin{aligned} f\ x\ z &= f\ z\ x \\ &= x \end{aligned}$$

- In terms of performance, foldl is typically more efficient due to constant stack space. But not always!

Flattening a List of Lists

- An example:
 $\text{concat } [[1, 2], [], [3]] \rightarrow [1, 2, 3]$
- Implement in terms of foldr.
$$\begin{aligned} \text{concat } \mathbf{xss} \\ = \text{foldr } (\mathbf{++}) \mathbf{[] } \mathbf{\mathbf{xss}} \end{aligned}$$
- Example execution:

```
concat [[1,2],[],[3]]
→ foldr (++) [] [[1,2],[],[3]]
→ aux [[1,2],[],[3]]
→ [1,2] ++ (aux [])
→ [1,2] ++ ([] ++ (aux [3]))
→ [1,2] ++ ([] ++ ([3] ++ (aux [])))
→ [1,2] ++ ([] ++ ([3] ++ []))
→ [1,2,3]
```

```
let foldr f z xs =
  let aux xs =
    case xs of
      [] -> z
      y:ys -> f y (aux ys)
  in aux xs
```

Flattening a List of Lists

- Implement in terms of fold_left.

```
concat XSS
= foldl (++) [] XSS
```

```
let foldl f z xs =
  let aux acc xs =
    match xs with
    [] -> acc
    y:ys -> aux (f y acc) ys
  in aux z xs
```

- Example execution:

```
concat [[1,2],[],[3]]
→ foldl (++) [] [[1,2],[],[3]]
→ aux [] [[1,2],[],[3]]
→ aux ([] ++ [1,2]) [[], [3]]
→ aux ((([] ++ [1,2]) ++ []) [[3]])
→ aux (((([] ++ [1,2]) ++ []) ++ [3])) []
→ ((([] ++ [1,2]) ++ []) ++ [3])
```

Fold Left versus Fold Right?

- Essential difference:

`x1 ++ (x2 ++ (x3 ++ (x4 ++ [])))`

versus

`((([] ++ x1) ++ x2) ++ x3) ++ x4`

- Which is better? Assume each of x1..x4 is 10 elements

`x1 ++ (x2 ++ (x3 ++ (x4 ++ [])))`

takes 10+10+10+10 steps

`([] ++ x1) ++ x2) ++ x3) ++ x4`

takes 0+10+20+30 steps

Foldable in Haskell

- Folding over List :

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

- Folding over Tree :

```
foldr :: (a->b->b) -> b > Tree a -> b
```

- Generic Folding :

```
foldr :: Foldable t => (a->b->b) -> b > t a -> b
```

```
class Foldable (t :: * -> *) where
  ...
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
```

Foldable Type Class

```
class Foldable (t :: * -> *) where
    :
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldl :: (b -> a -> b) -> b -> t a -> b
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
    null :: t a -> Bool
    length :: t a -> Int
    elem :: Eq a => a -> t a -> Bool
    maximum :: Ord a => t a -> a
    minimum :: Ord a => t a -> a
    sum :: Num a => t a -> a
    product :: Num a => t a -> a
```

Examples of Foldable

```
instance Foldable [] -- Defined in 'Data.Foldable'  
instance Foldable Maybe -- Defined in 'Data.Foldable'  
instance Foldable (Either a) -- Defined in 'Data.Foldable'  
instance Foldable ((,) a) -- Defined in 'Data.Foldable'
```

- Guess of output of these executions:

```
let foo = foldr (\x y -> x+y) 0  
foo [1..4]  
foo (Just 10)  
foo Nothing
```

List Mapping

- Transform one list into another :

```
map f xs =
  let aux xs =
    case xs of
      [] -> []
      y:ys -> (f y) :(aux ys)
  in aux xs
```

- Usages :

```
let double xs = map (fun x -> 2*x) xs
```

```
let is_pos xs = map (fun x -> x>0) xs
```

Type of Map

- Map function :

```
map f xs =  
let aux xs =  
  case xs of  
    [] -> []  
    y:ys -> (f y) :(aux ys)  
in aux xs
```

- Type of map :

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

- Note that result type of list is changed by function parameter of type $(a \rightarrow b)$.

Example Execution

```
map f xs =
  let aux xs =
    case xs of
      [] -> []
      y:ys -> (f y) : (aux ys)
  in aux xs
```

Example :

```
double [1,2]
→ map (fun x -> 2*x) [1,2]
→ aux [1,2]
→ ((fun x -> 2*x) 1) : aux [2]
→ ((fun x -> 2*x) 1) : (fun x -> 2*x) 2) : aux []
→ 2 : 4 : []
→ [2,4]
```

Example Execution

```
map f xs =  
  let aux xs =  
    case xs of  
      [] -> []  
      y:ys -> (f y) : (aux ys)  
  in aux xs
```

Example :

```
is_pos [1,-2]  
→ map (fun x -> x>0) [1,-2]  
→ aux [1,-2]  
→ ((fun x -> x>0) 1) : aux [-2]  
→ ((fun x -> x>0) 1) : (fun x -> x>0) -2) : aux []  
→ True : False : []  
→ [True, False]
```

Functor in Haskell

- Mapping over List :

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

- Mapping over Tree :

```
map :: (a -> b) -> Tree a -> Tree b
```

- Generic Mapping :

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

- Can you implement (`<$`) in terms of `fmap` :

Examples of Functors

```
instance Functor (Either a) -- Defined in 'Data.Either'  
instance Functor [] -- Defined in 'GHC.Base'  
instance Functor Maybe -- Defined in 'GHC.Base'  
instance Functor IO -- Defined in 'GHC.Base'  
instance Functor ((->) r) -- Defined in 'GHC.Base'  
instance Functor ((,) a) -- Defined in 'GHC.Base'
```

- Guess of output of these executions:

```
let foo = fmap (\x -> x+1)  
foo [1..10]  
foo (Just 10)  
foo Nothing  
foo (\x -> x*x) 3
```

Composition

- We can write a general compose operator:

```
let compose g f x = g (f x)
```

- Type of compose :

```
compose :: (b -> c) -> (a -> b) -> a -> c
```

- Similar to Unix pipe :

```
cat x | f | g
```

Except Unix pipe was for a text file, and presented in infix notation.

Infix Version of Composition

- In Haskell, we use an infix version of `compose`

```
let (.) g f x = g (f x)
```

- Type of `compose` :

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- How is `compose` related to `fmap` ?

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Unix Pipe in Haskell



- Declare an infix pipe operator:

```
let (|>) :: a -> (a->b) -> b
    a |> f = f x
```

- Example of use:

```
x
|> f
|> g
```

Equivalent to:

```
(x |> f) |> g = g (f x)
```

An Example of Pipe

- Let us first declare:

```
let double xs = map (fun x -> 2*x) xs
```

```
let sum xs = foldl (+) 0 xs
```

- Example of use:

```
[1,2,3]
| > double
| > double
| > sum
```

Weak Precedence Apply Operator

- Another infix apply operator:

```
(\$) :: (a->b) -> a -> b  
f \$ x = f x
```

- `\$` is essentially function apply but with very weak precedence:

- Example of use:

```
inc \$ x*2  
= inc (x*2)
```

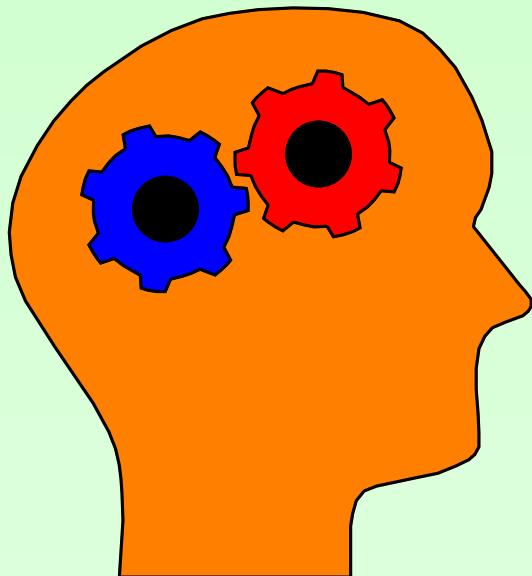
- Without `\$`, the default application gives:

```
inc x*2  
= (inc x)*2
```



CS2104: Programming Languages Concepts

Lecture 5 : Other Haskell Features



*“Comprehension Syntax and
More Type-Classes”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Haskell – Other Highlights

- List Comprehension
- Numbers
- Arrays
- Monoid Type Class

Comprehension

Syntactic sugar makes codes more readable

List Comprehension

- List comprehension is a useful *shorthand* for building list data structures:

`[f x | x <- xs]`

- Captures a list of all `f x` where `x` is drawn from `xs`.
More than one generators are allowed:

`[(x,y) | x <- xs, y <- ys]`

- Guards are also permitted. Example :

```
quicksort []      = []
quicksort (x:xs) = quicksort [y | y <- xs, y < x]
                  ++
                  [x]
                  ++
                  quicksort [y | y <- xs, y >= x]
```

Translation for List Comprehension

- Given:

$[f x \mid x \leftarrow xs]$

- This get translated to:

`map (\ x -> f x) xs`

- Recall:

`map f [] = []`

`map f (x:xs) = (f x):(map f xs)`

Translation for List Comprehension

- Given:

$[f x \mid x \leftarrow xs, x > 5]$

- This get translated to:

`map (\x -> f x) (filter (\x -> x > 5) xs)`

- Recall:

`filter f [] = []`

`filter f (x:xs) = if (f x) then x : (filter f xs)
else filter f xs`

Translation for List Comprehension

- Given:

$[(x,y) \mid x \leftarrow xs, y \leftarrow ys]$

- This get translated to:

`concatMap (\ x -> map (\y -> (x,y)) ys) xs`

where:

`concatMap f [] = []`

`concatMap f (x:xs) = (f x) ++ (concatMap f xs)`

Translation for List Comprehension

- General translation scheme:

$$\begin{aligned} [e \mid x \leftarrow xs] \\ \rightarrow \text{map } (\lambda x \rightarrow e) xs \end{aligned}$$
$$\begin{aligned} [e \mid x \leftarrow xs, y \leftarrow ys, \text{rest}] \\ \rightarrow \text{concatMap } (\lambda x \rightarrow [e \mid y \leftarrow ys, \text{rest}]) xs \end{aligned}$$
$$\begin{aligned} [e \mid x \leftarrow xs, \text{test}, \text{rest}] \\ \rightarrow [e \mid x \leftarrow \text{filter } (\lambda x \rightarrow \text{test}) xs, \text{rest}] \end{aligned}$$

Exercise

[e | x <- xs] → map (\x -> e) xs

[e | x <- xs, y <- ys, rest] → concatMap (\x -> [e | y <- ys, rest]) xs

[e | x <- xs, test, rest] → [e | x <- filter (\x -> test) xs, rest]

[(x+x,j) | x <- [1..3], x < 2, j <- [7..8]] →

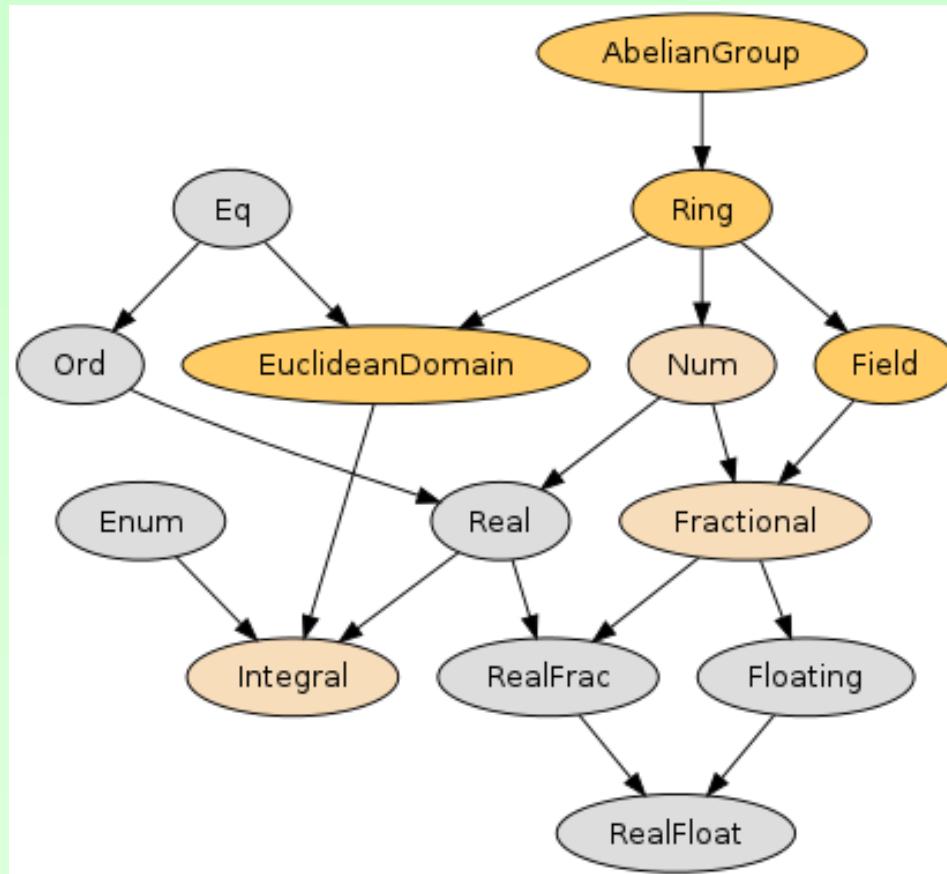
More Type Classes

Recap of Type Classes

- Supports systematic overloading via types.
- So far, mostly first-order type class
- Type sub-classes can be supported (rich hierarchy)
- Captures only type signatures but not properties, such as associativity and commutativity etc

Numbers

Hierarchy in Prelude YAP



Numbers

- Num class has the following basic operations: ...

```
class Num a where
    (+), (-), (*)   :: a -> a -> a
    negate, abs     :: a -> a
```

- Division via div/mod is supported in Integral class, while (/) is supported by Fractional class.
- The Floating class contains trigonometric, logarithmic and exponential functions.

Constructed Numbers

- Standard numeric types `Int`, `Integer`, `Float`, `Double` are primitives and other numeric types are constructed from these
- `Complex` type is under `Floating` but made from `Realfloat`, as follows:

```
data (realFloat a) => Complex a
    = !a :+ !a deriving (Eq, text)
```

```
conjugate :: RealFloat a => Complex a -> Complex a
conjugate (x :+ y) = x :+ (-y)
```

Constructed Numbers

- Another class is `Ratio` :

```
(%)           :: (Integral a) => a => a -> Ratio a
numerator, denominator
                  :: Integral a => Ratio a -> a
```

- This is quite different from `Complex`

Arrays

Arrays

- Arrays can be regarded as functions from indices to values, but for efficient retrieval it has to be contiguous and bounded.

```
class (Ord a) => Ix a where
    range      :: (a,a) -> [a]
    index      :: (a,a) -> a -> Int
    inRange    :: (a,a) -> a -> Bool
```

- Possible index types : `Int`, `Integer`, `Char`, `Bool`, tuples of `Ix` type upto length 5
- Possible bounds:
 - (0,9) for 1-dimensional array with 10 elements
 - ((1,1),(100,100)) for 2-dimensional array

Arrays

- `range` enumerates a list of indices in index order.

```
range (0,4)          ⇒      [0,1,2,3,4]
range ((0,0), (1,2)) ⇒      [(0,0), (0,1), (0,2), (1,0),
                               (1,1), (1,2)]
```

- `inRange` checks if an index is between a pair of bounds
- `index` calculates zero-origin offset of an index from its bounds

```
index (1,9) 2          ⇒      1
index ((0,0), (1,2)) (1,1) ⇒      4
```

Array Creation

- Can build an array from an association list with:

```
array  ::  (Ix a)  => (a,a)  -> [(a,b)]  -> Array a b
```

- An array of squares from 1 to 100

```
squares = array (1,100) [(i,i*i) | i <- [1..100]]
```

- Array subscription done with ! Operator.

```
squares ! 7          ⇒      49
```

- Bounds of an array determined by

```
bounds squares       ⇒      (1,100)
```

Recursive Array

- Arrays may be defined recursively

```
fibs    :: Int -> Array Int Integer
fibs n = a where a = array (0,n) [(0,1),(1,1)] ++
              [(i, a!(i-2)+a!(i-1)) | i <- [2..n]]
```

- Lazy evaluation avoids need to worry on the order of evaluating such recursive arrays. A wavefront example:

```
wavefront :: Int -> Array (Int,Int) Int
wavefront n = a where
    a = array ((1,1), (n,n))
        (((1,j),1) | j <- [1..n]) ++
        (((i,1),1) | i <- [2..n]) ++
        (((i,j),a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j)))
            | i <- [2..n], j <- [2..n])
```

Accumulation

- While index is unique for array creation, we may wish to accumulate a number of values into each index.

accumulating function initial value bounds
elements to accumulate

```
accumArray :: (Ix a) => (b->c->b) -> b -> (a,a)
              -> [Assoc a b] -> Array a b
```

• histogram can calculate occurrences

```
hist :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i,1) | i <- is,
                                         inRange bnds i]
```

```
e17 :: Array Char Int
e17 = hist ('a','z') "This counts the frequencies of each lowercase
letter"
```

Incremental Updates

- An incremental update operation allows a modified array to be returned, e.g. `a // [(i,v), (j,w)]`

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

- Its index may appear multiple times with the last value taking precedence.
- An example to swap two rows.

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c
           -> Array (a,b) c
swapRows i i' a = a // [assoc | j <- [jLo..jHi],
                                assoc <- [((i,j),a!(i',j)),
                                           ((i',j),a!(i,j))]]
where ((iLo,jLo),(iHi,jHi)) = bounds a
```

Algebraic Structures

Semi-Group and Monoids

- We can capture *mathematical* structures as type classes.

```
class SemiGroup a where
    op          :: a -> a -> a
class SemiGroup a => Monoid a where
    unit        :: a
```

- Above declaration only captures the type signatures.
Two properties of monoids are:

```
unit `op` x = x `op` unit = x
(x `op` y) `op` z = x `op` (y `op` z)
```

- But these properties are important but not checked by Haskell. We assume that users ensure them.

Semi-Group and Monoids

```
class SemiGroup a where
    op          :: a -> a -> a
class SemiGroup a => Monoid a where
    unit        :: a
```

- Above declaration only captures the type signatures.
Two properties of monoids are:

```
unit `op` x = x `op` unit = x
(x `op` y) `op` z = x `op` (y `op` z)
```

- But these properties are important but not checked by Haskell. We assume that users ensure them.

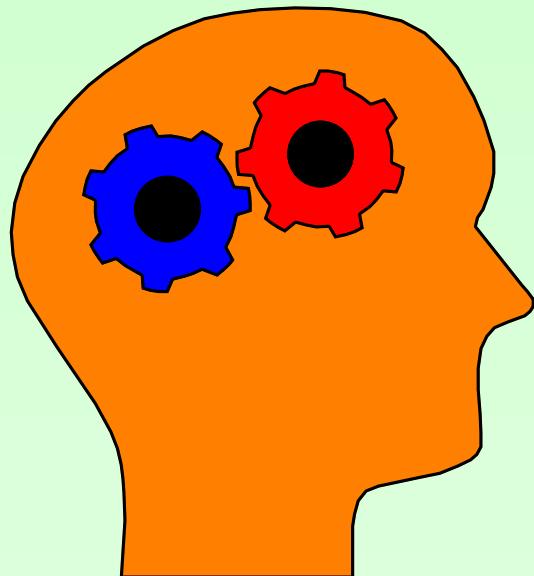
Higher-Order Type Classes

Monoid \rightarrow *Monads*



CS2104: Programming Languages Concepts

Lecture 6 : Towards Monads



*“Imperative Programming
in a Pure Language”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Can be challenging but You are Not Alone

- The midnight Monad, a journey to enlightenment.

<https://www.lambdacat.com/the-midnight-monad-a-journey-to-enlightenment/>

- Functors, Applicatives and Monads in Picture form:

http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

- A Fistful of Monads.

<http://learnyouahaskell.com/a-fistful-of-monads>

Pure vs Impure Code

- Imperative Programming (with side effects).

```
print :: String -> ()
```

```
let c = print("hello")
in c ; c
```

```
print("hello");
print("hello")
```

different

- Pure Monadic Programming.

```
print :: String -> IO ()
```

```
let c = print("hello")
in c >> c
```

```
print("hello") >>
print("hello")
```

equivalent

Pure Value World

Here's a simple value:

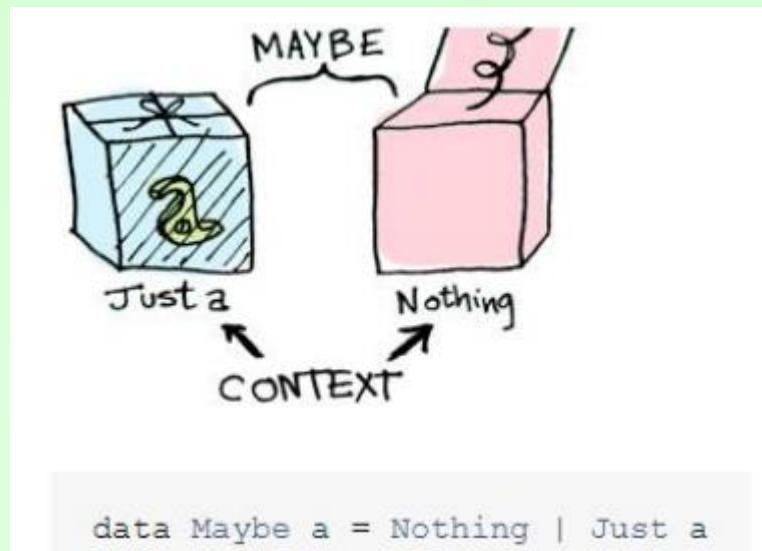
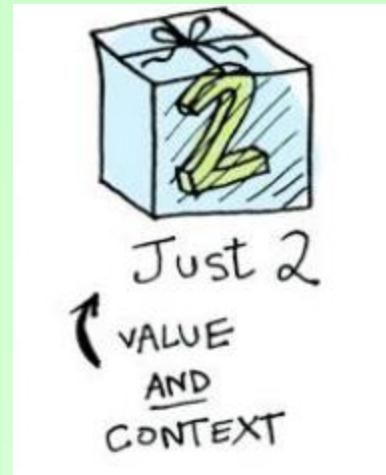


And we know how to apply a function to this value:



Value within Some Context

- Value and a Context.
- Maybe Type where
where
Nothing
denotes error

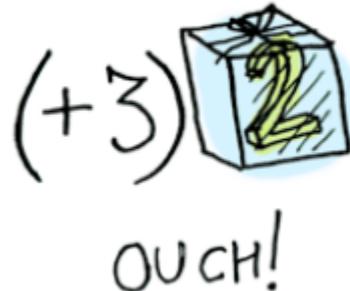


Other Examples of Context

- **[a]**
 - for non-determinism
- **state → (state, a)**
 - for imperative state that can be updated
- **Parser a = String → [(a, String)]**
 - For non-deterministic parsing
- **IO a**
 - for input-output interaction

Why Functor?

When a value is wrapped in a context, you can't apply a normal function to it:



- Solution : Functor.

```
> fmap (+3) (Just 2)  
Just 5
```



What is a Functor?

Functor is a typeclass. Here's the definition:

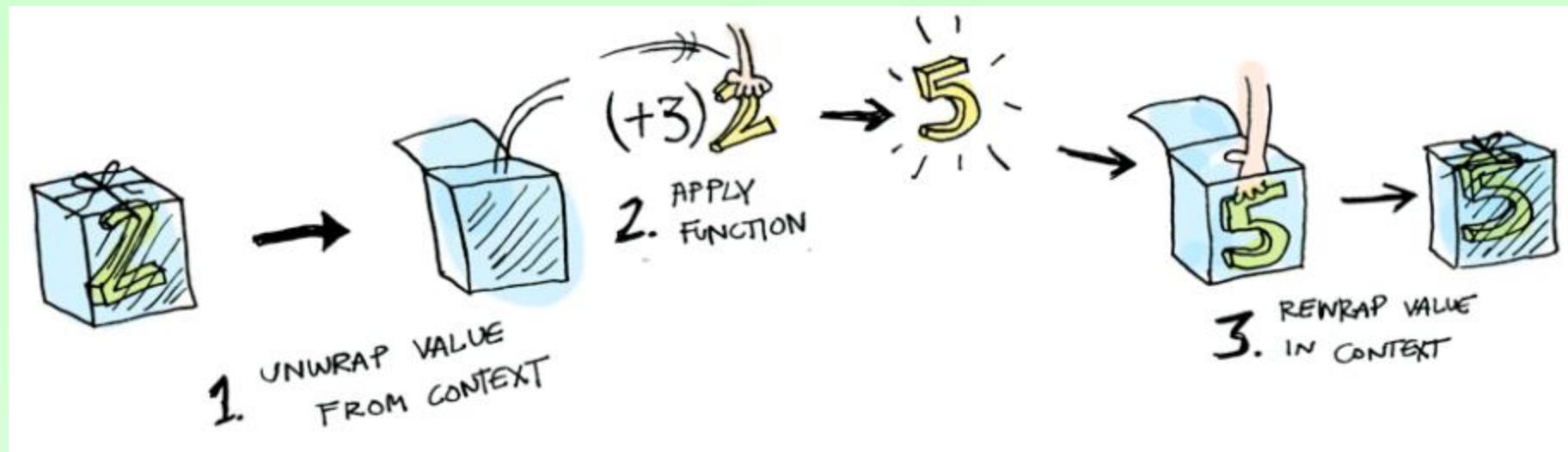
1. TO MAKE A DATA TYPE f
A FUNCTOR,

class Functor f where

\rightsquigarrow $fmap :: (a \rightarrow b) \rightarrow fa \rightarrow fb$

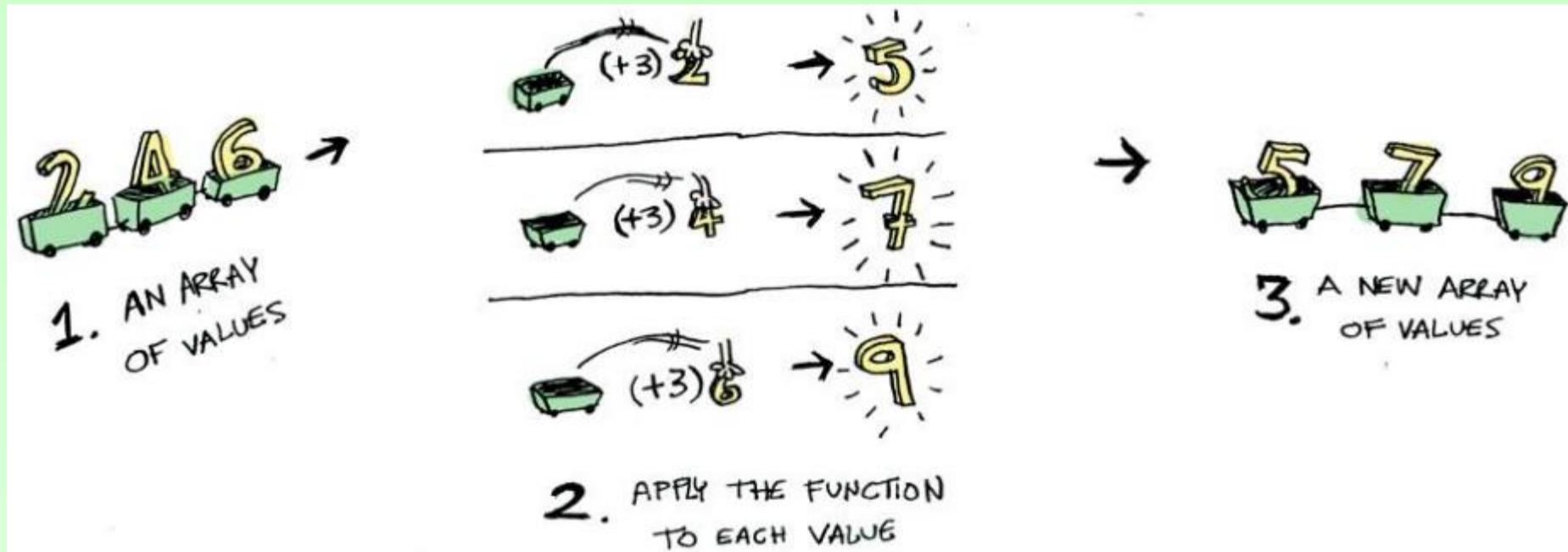
2. THAT DATA TYPE
NEEDS TO DEFINE
HOW $fmap$ WILL
WORK WITH IT.

Behind the Scene



```
instance Functor Maybe where
    fmap func (Just val) = Just (func val)
    fmap func Nothing = Nothing
```

List/Arrays are also Functors



fmap (+3) [2, 4, 6] ➔ [5, 7, 9]

(+3) <\$> [2, 4, 6] ➔ [5, 7, 9]

↑
infix variant

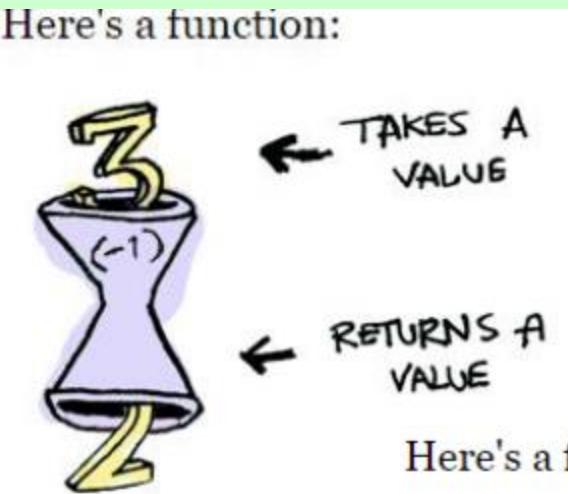
List as Functors

```
instance Functor [] where  
    fmap = map
```

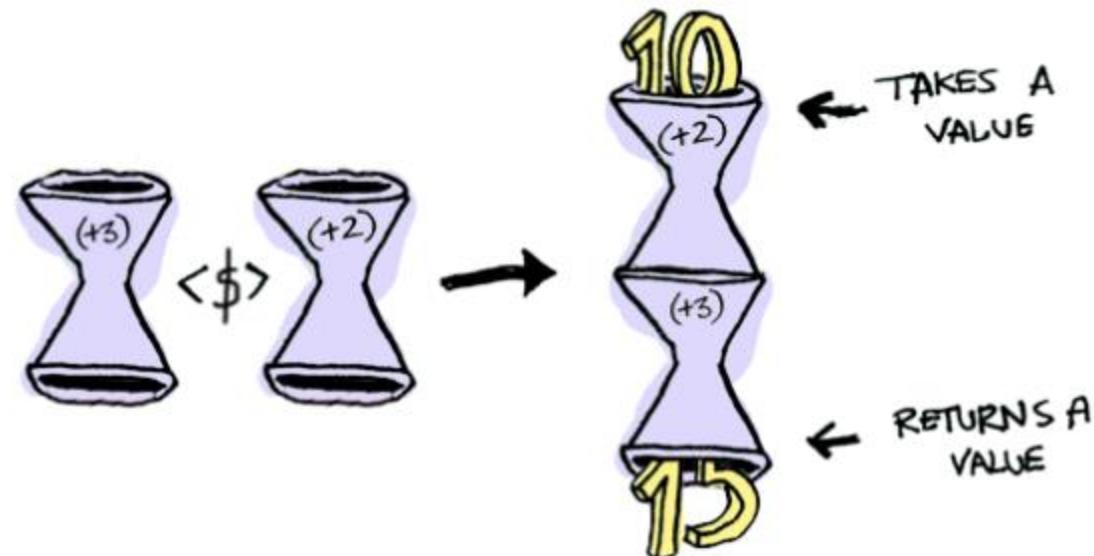
- List denotes non-determinism
- Examples:
 - `[]` means no solution
 - `[r1, r2, r3]` means three possible solutions

Functions are also Functors

Here's a function:



Here's a function applied to another function:



Functions as Functors ..

```
> let foo = fmap (+3) (+2)  
> foo 10  
15
```

- Implementation

```
instance Functor ((->) r) where  
    fmap f g = f . g
```

What IF Functions are Wrapped in Context?



`Maybe (Int -> Int)`

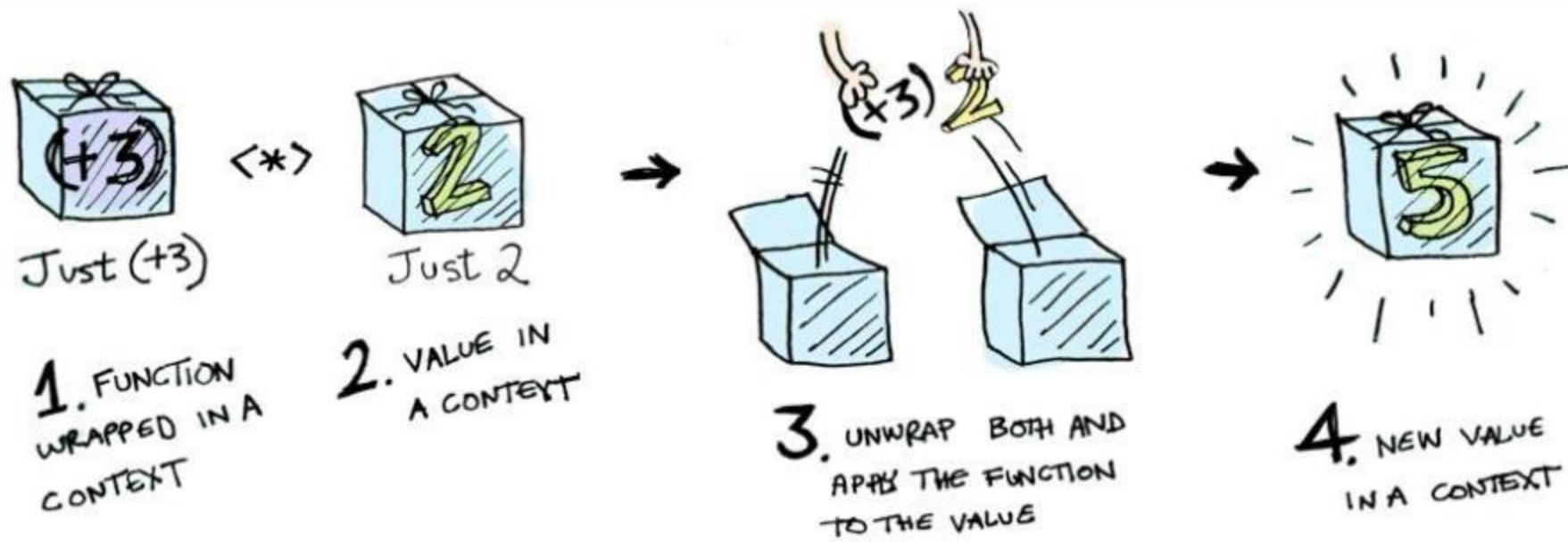


`Maybe (Int)`

Cannot use `fmap`

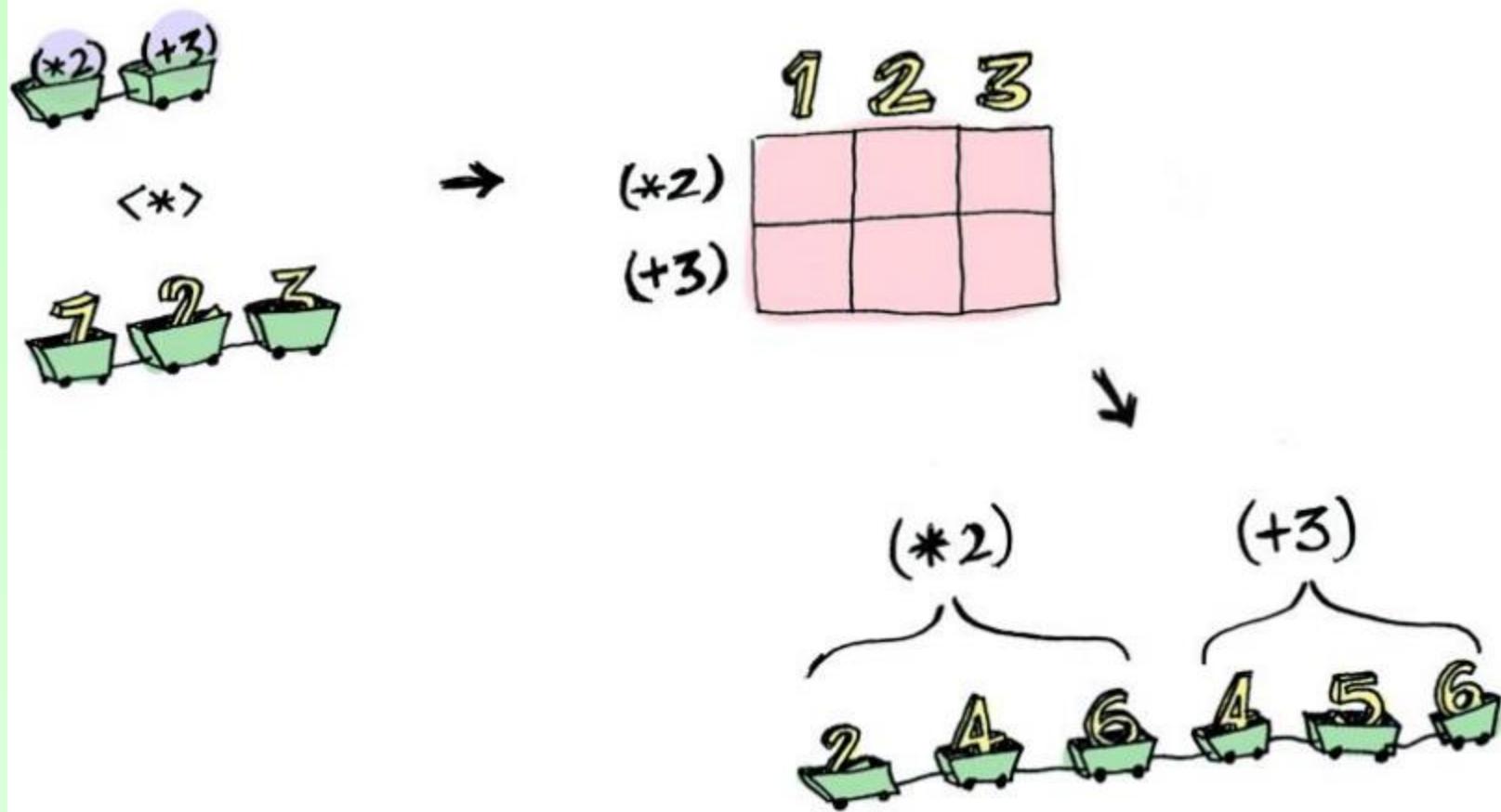
$$fmap ::= (\underset{\nearrow}{a} \rightarrow b) \rightarrow f_a \rightarrow f_b$$

Applicative to the Rescue



```
Just (+3) <*> Just 2 == Just 5
```

Applicative in List Context



```
> [(\ast 2), (+3)] <*> [1, 2, 3]  
[2, 4, 6, 4, 5, 6]
```

Why do we Need Applicative?

- Applicative can work with functions of any no. of arguments
 - Use fmap first

```
> let f = fmap (+) [1,2,3]
➤ :t f
➤ f :: Num a => [(a -> a)]
```

- Use Applicative now

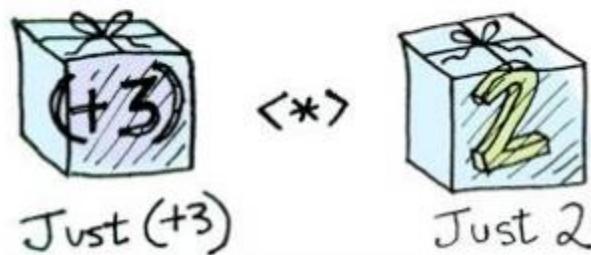
```
> f <*> [4,5]
➤ => [5,6,6,7,7,8]
```

Recap

Functors apply a function to a wrapped value:



Applicatives apply a wrapped function to a wrapped value:

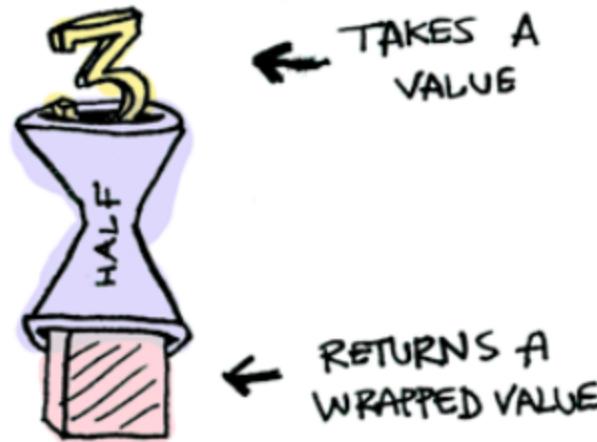


Essence of Monads

- How do we supply a wrapped value ($M\ a$) to a function which returns a wrapped value ($a \rightarrow M\ b$)
half :: Int -> Maybe Int

Suppose `half` is a function that only works on even numbers:

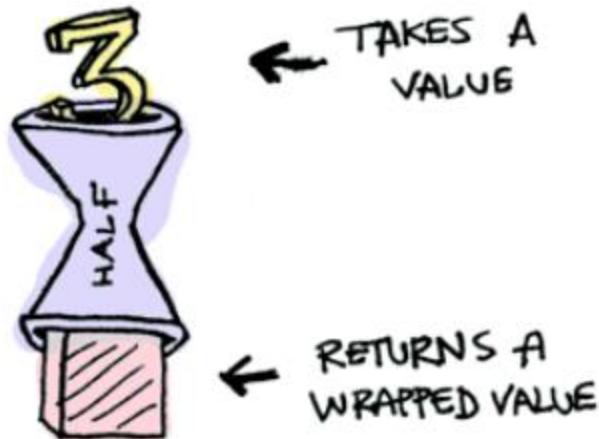
```
half x = if even x
          then Just (x `div` 2)
          else Nothing
```



What if we Apply on a Wrapped Value?

Suppose `half` is a function that only works on even numbers:

```
half x = if even x  
          then Just (x `div` 2)  
          else Nothing
```



Monad as a Type Class

```
class Monad m where  
  (">>=) :: m a -> (a -> m b) -> m b
```

Where `>=` is:

$(\gg=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b$

1. $\gg=$ TAKES A MONAD (LIKE `Just 3`)

2. AND A FUNCTION THAT RETURNS A MONAD (LIKE `half`)

3. AND IT RETURNS A MONAD

Chaining via Monads

```
> Just 20 >>= half >>= half >>= half  
Nothing
```

```
instance Monad Maybe where  
    Nothing >>= func = Nothing  
    Just val >>= func = func val
```



Input-Output as a Monad



*getLine takes no arguments
and gets user input.*



IO Monad Operation

`readFile` takes a string (a filename) and returns that file's contents



```
readFile :: FilePath -> IO String
```

IO Monad Operation

`putStrLn` takes a string and prints it:



`putStrLn`

```
putStrLn :: String -> IO ()
```

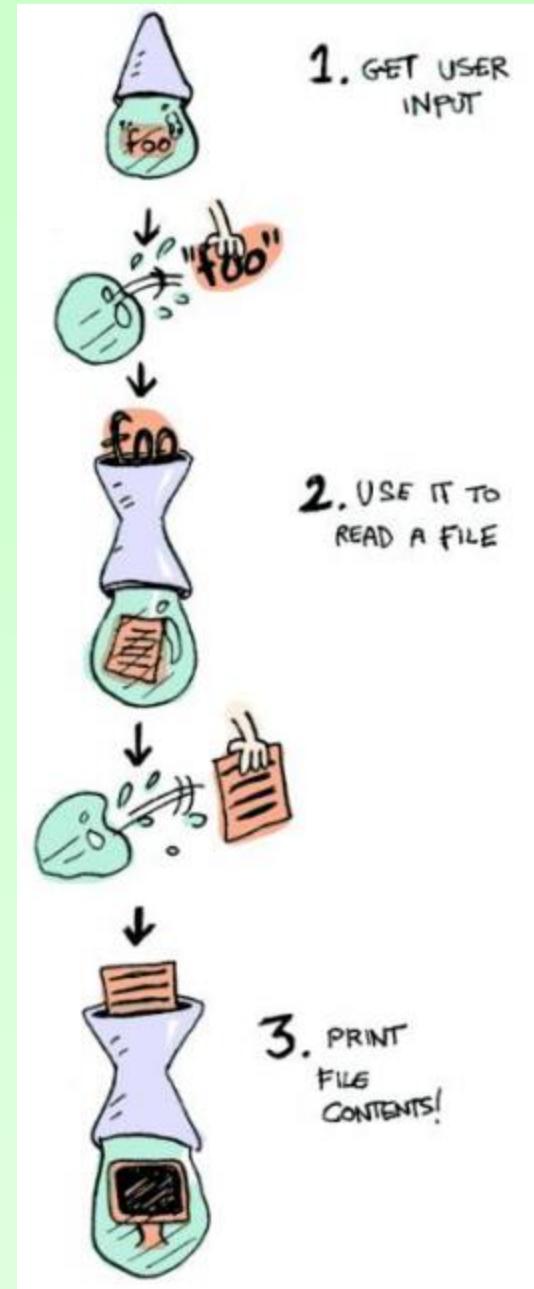
Chaining IO Operation

Chaining

```
getLine >>= readFile >>= putStrLn
```

Syntactic Sugar

```
foo = do
    filename <- getLine
    contents <- readFile filename
    putStrLn contents
```

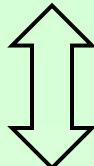


Do Comprehension

- Syntactic sugar notation for Monads.
- List is an instance of monad, and list comprehension is an instance of Do-comprehension!

```
[ (x,y) | x <- xs, test x, y<-ys]
```

```
do  
  x <- xs  
  filter test  
  y <- ys  
  return (a,b)
```



```
filter test = \ x ->  
             if test x then return a else empty
```

Monads

Monads Formally

- Another example of higher-order type class is:

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

    >> :: (m a) -> (m b) -> m b
    m1 >> m2      = m1 >>= (\_ -> m2)
```

- Laws of `Monad` class:

$$\begin{aligned} (\text{return } a) >>= k &= k \ a \\ m >>= \text{return} &= m \\ (m >>= (\lambda a \rightarrow (k \ a) >>= (\lambda b \rightarrow h \ b))) \\ &= (m >>= (\lambda a \rightarrow k \ a) >>= (\lambda b \rightarrow h \ b)) \end{aligned}$$

- IO is an instance of Monad ...

Input/Output

Input/Output

- The I/O system in Haskell is purely functional but has all the expressive power of conventional imperative languages.
- Actions are *defined* rather than *invoked* in an expression-oriented style.
- These actions are modelled as *monads* of type `IO t` which is a conceptual structure with some properties that supports imperative actions.

Basic I/O Operations

- Every I/O action returns a value, e.g :

```
getChar      :: IO Char
```

- Some IO actions also take input(s)

```
putChar      :: Char -> IO ()
```

- `IO` is an instance of the the `Monad` class.
- Actions are sequenced by bind operator:

```
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
(>>)   :: Monad m => m a -> m b -> m b
```

Basic I/O Operations

- The do statement captures a sequence of actions, e.g :

```
main    :: IO ()  
main    = do c <- getChar  
            putChar c
```

- Syntactic sugar for the following:

```
main    = getChar >>=  
        (\ c -> putChar c)
```

- How to return a value from sequence of actions?.

```
ready  = IO Bool  
ready  = do c <- getChar  
            c == 'y'  
            ↘  
            return (c == 'y')
```

Bigger I/O Operations

- Function to get a string of char may use recursion, as follows:

```
getLine :: IO String
getLine = do c <- getChar
            if c=='\n' then return ""
            else do l <- getLine
                    return (c:l)
```

- A pure value can be converted into an action by return, but the not the converse. Illegal to use:

```
x + print y
```

- Function `f::Int -> Int -> Int` cannot do any IO at all, unless we make use of unsafe operations.

Building Actions

- IO operations are ordinary Haskell values that can be passed to functions, placed into data structures and returned as results etc.
- Example : we can build a list of actions.

```
todoList :: [IO ()]
todoList = [ putChar 'a',
            do {putChar 'b'; putChar 'c'},
            do {x <- getChar; putChar x} ]
```

- Can combine them into a single action using:

```
sequence_ :: [IO ()] -> IO ()
sequence_ = foldr (">>>") (return ())
```

Imperative Programming

- I/O programming in Haskell is very close to that being done for ordinary imperative programming.
- As a comparison, imperative `getLine` is simply:

```
function getLine() {  
    c := getChar();  
    if c=='\n' then return ""  
    else {l:=getLine();  
          return c:l} }
```

- Main difference is that no special semantics is needed and the entire code is still purely functional. Monad cleanly separates the pure from imperative.

Recap / Comparison

- Imperative `getLine` in C:

```
function getLine() {  
    c := getChar();  
    if c=='\n' then return ""  
    else  {l:=getLine();  
           return c:l} }
```

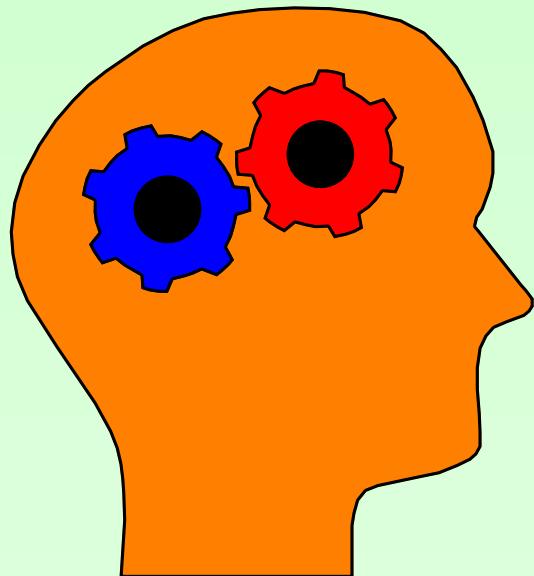
- Monadic IO in Haskell

```
getLine  :: IO String  
getLine  = do c <- getChar  
             if c=='\n' then return ""  
             else do l <- getLine  
                      return (c:l)
```



CS2104: Programming Languages Concepts

Lecture 7 : Lexical and Syntax Analysis



*“Monadic Parsing
with Haskell”*

Lecturer : Chin Wei Ngan
Email : chinwn@comp.nus.edu.sg

Language Syntax

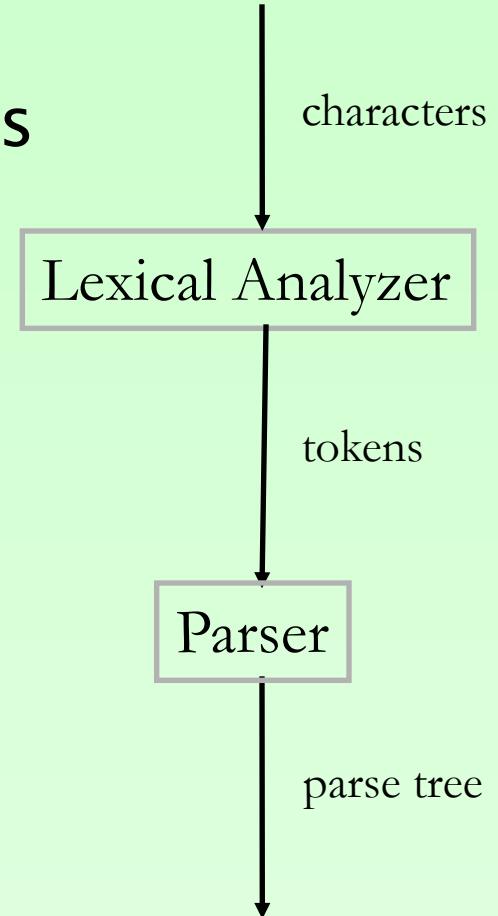
- Language = Syntax + Semantics
- The *syntax* of a language is concerned with the *form* of a program: how expressions, types, declarations and commands are formed.
- The *semantics* of a language is concerned with the *meaning* of a program : how programs should behave when executed on computers.

Language Syntax

- Defined by *grammar rules*
 - define how to make ‘expressions’ out of ‘words’
- For programming languages
 - sentences are called *expressions*
 - words are called *tokens*
 - grammar rules describe both tokens and statements

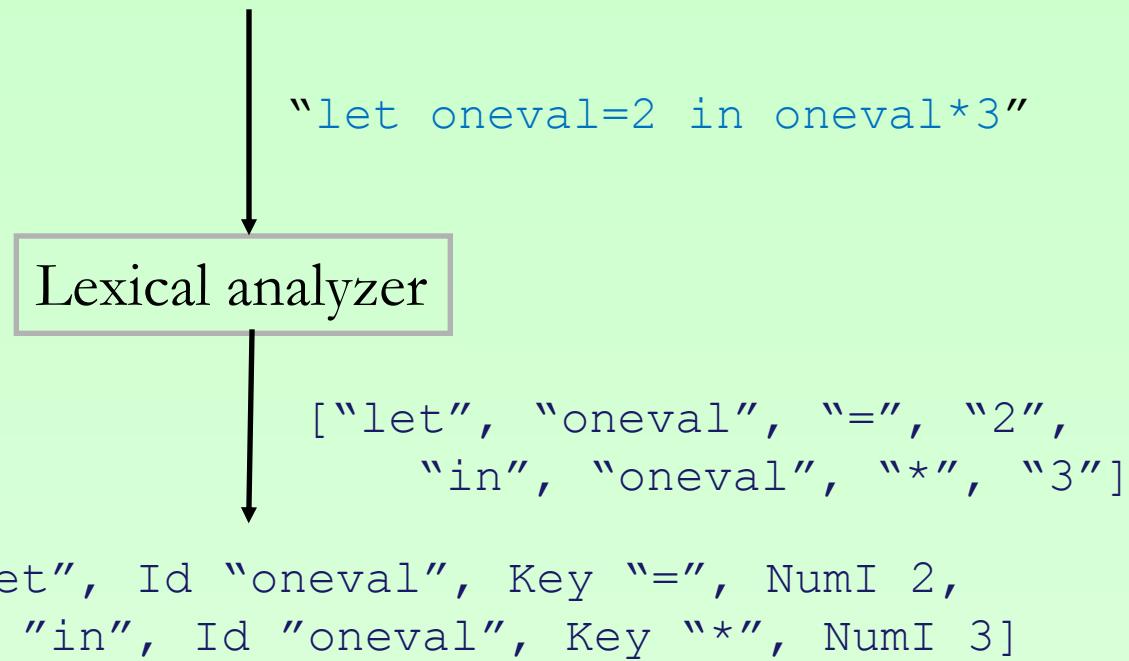
Language Syntax

- *Token* is sequence of characters
- *Expression* is sequence of tokens
- *Lexical analyzer* is a program
 - recognizes character sequence
 - produces token sequence
- *Parser* is a program
 - recognizes a token sequence
 - produces statement representation
- Statements are represented as *parse trees*



Lexical Analysis

- *Example:*



- Categorizing the tokens help with parsing.

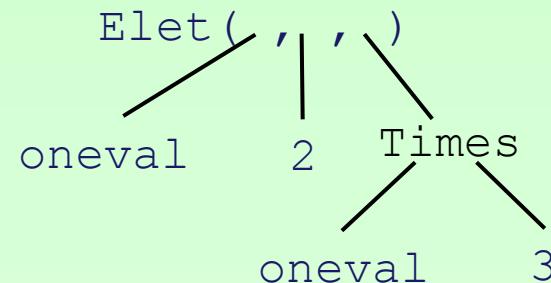
```
data Token = Key String | Id String | NumI Int
```

Syntax Analysis

- Example:

[Key "let", Id "oneval", Key "=",
NumI 2, Key "in", Id "oneval",
Key "*", NumI 3]

Syntax Analysis



- We can define a parse tree using:

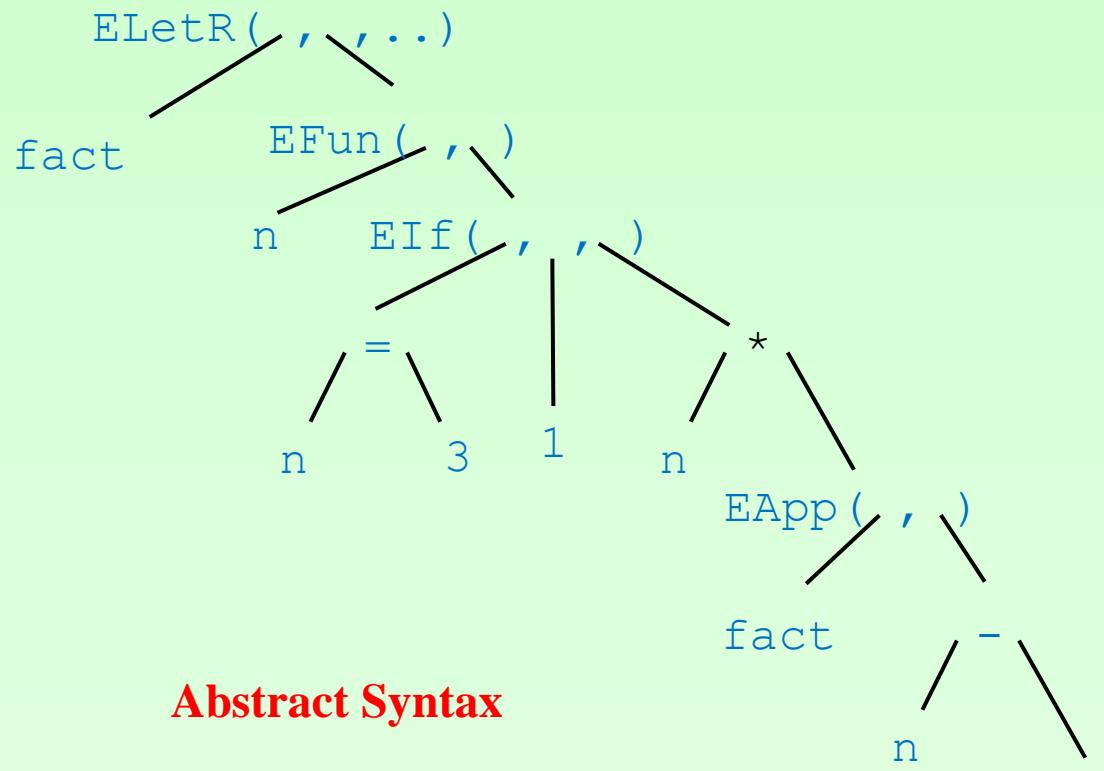
```
data Exp = Var String
    | Elet String Exp Exp
    | Enum Int
    | Times Exp Exp
    | ...
```

Abstract Syntax Tree

- Used to represent code fragment as a data structure.
- Below is an example for recursive function definition in Haskell

```
let fact =  
  (\ n ->  
    if n = 0  
      then 1  
    else n*  
      (fact(n-1))  
  )
```

Concrete Syntax



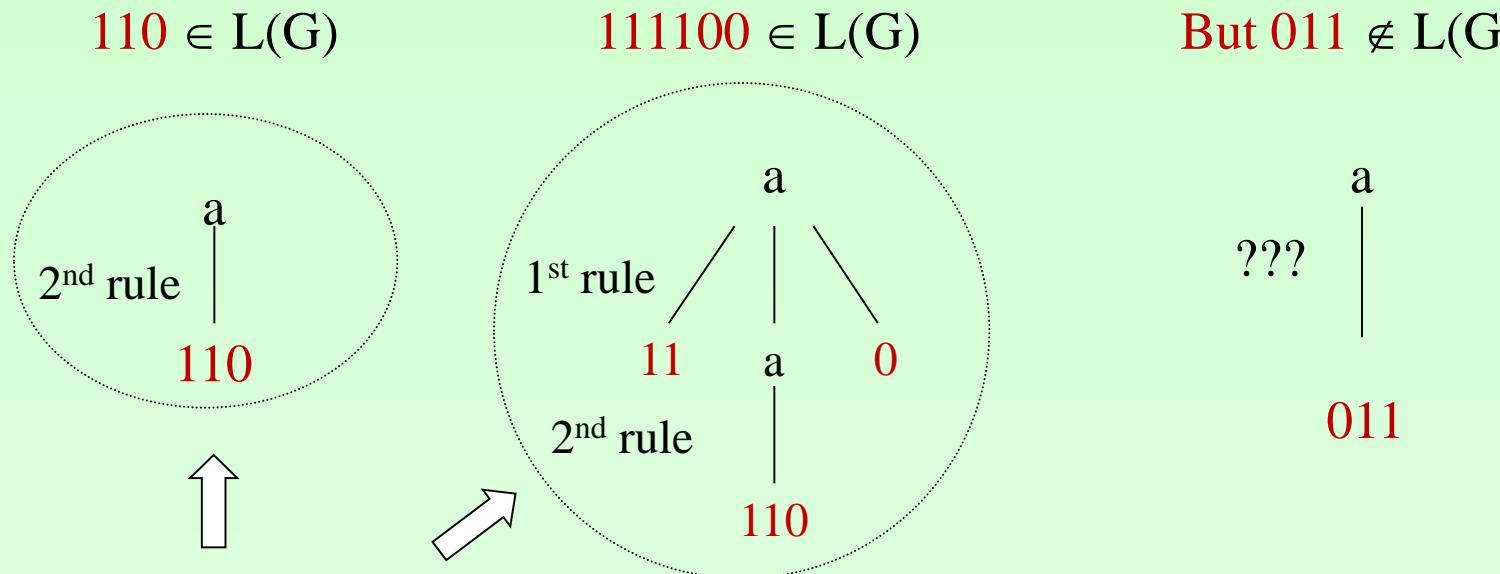
Abstract Syntax

Context-Free Grammars

- A context-free grammar (CFG) is:
 - A set of **terminal** symbols T (tokens or constants)
 - A set of **non-terminal** symbols N
 - One (non-terminal) **start symbol** σ
 - A set of **grammar (rewriting) rules** Ω of the form
 $\langle \text{nonterminal} \rangle ::= \langle \text{sequence of terminals and nonterminals} \rangle$
- Grammar rules (productions) can be used to
 - verify that an expression is legal
 - generate all possible statements
- The set of all possible statements generated by a grammar from the start symbol is called a (*formal language*)

An Example

- Let $N = \{\langle a \rangle\}$, $T = \{0,1\}$, $\sigma = \langle a \rangle$
 $\Omega = \{\langle a \rangle ::= 11\langle a \rangle 0, \langle a \rangle ::= 110\}$
-



These trees are called *parse trees* or *syntax trees* or *derivation trees*.

Why CFG?

- A programming language may have arbitrary number of nested statements, such as:
if-then-else, let---in-, and so on.
- $L_1 = \{(\text{if-then})^n (\text{let-in})^m \mid n, m > 0\}$
- These can be easily supported by CFG.

Backus-Naur Form

- BNF is a common notation to define context-free grammars for programming languages
- $\langle \text{digit} \rangle$ is defined to represent one of the ten tokens 0, 1, ..., 9

$$\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9$$

- (Positive) Integers

$$\langle \text{integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{integer} \rangle$$
$$\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9$$

- $\langle \text{integer} \rangle$ is defined as the sequence of a $\langle \text{digit} \rangle$ followed by zero or more $\langle \text{digit} \rangle$'s

Extended Backus-Naur Form

- EBNF is a more compact notation to define the syntax of programming languages.
- EBNF has the same power as CFG.
- It provided some *shortcuts* to denote repetition.

- With EBNF, (positive) integers may be defined as:
$$\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$$
- Recall that BNF was:
$$\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$$

Notations

- $\langle x \rangle$ nonterminal x
- $\langle x \rangle ::= Body$ $\langle x \rangle$ is defined by $Body$
- $\langle x \rangle \mid \langle y \rangle$ either $\langle x \rangle$ or $\langle y \rangle$ (choice)
- $\langle x \rangle \langle y \rangle$ the sequence $\langle x \rangle$ followed by $\langle y \rangle$
- $\{ \langle x \rangle \}$ sequence of zero or more occurrences of $\langle x \rangle$
- $\{ \langle x \rangle \}^+$ sequence of *one or more* occurrences of $\langle x \rangle$
- $[\langle x \rangle]$ *zero or one* occurrence of $\langle x \rangle$

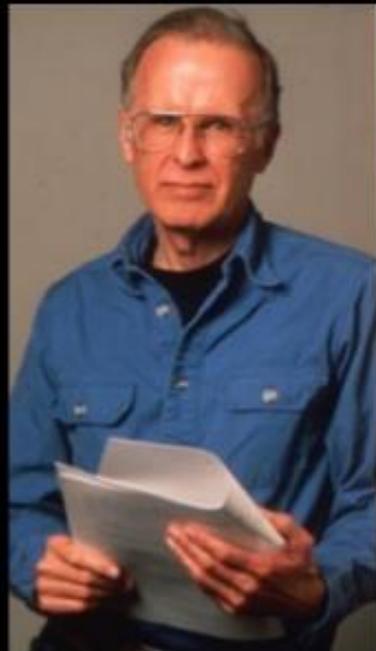
Our Pioneers

FORTRAN, BNF, FP

John Backus

Turing Award 1977

For profound, influential, and lasting contributions to the design of practical high level programming systems, notably through his work on FORTRAN, and for seminal publication of formal procedures for the specification of programming languages.



**Turing Centenary Celebrations
Persistent Systems Ltd.**

February 2013

abhijatv@gmail.com

EBNF for Haskell Fragment

$\langle \text{expression} \rangle ::=$

$\langle \text{lowercase-identifier} \rangle$

| $\langle \text{constants} \rangle$

| $\text{let } \langle \text{pattern} \rangle = \langle \text{expression} \rangle \text{ in } \langle \text{expression} \rangle$

| $\text{if } \langle \text{expression} \rangle \text{ then } \langle \text{expression} \rangle \text{ else } \langle \text{expression} \rangle$

| $\backslash \{ \langle \text{pattern} \rangle \}^+ \rightarrow \langle \text{expression} \rangle$

| $\langle \text{expression} \rangle \{ \langle \text{expression} \rangle \}^+$

| $(\langle \text{expression} \rangle) \mid \dots$

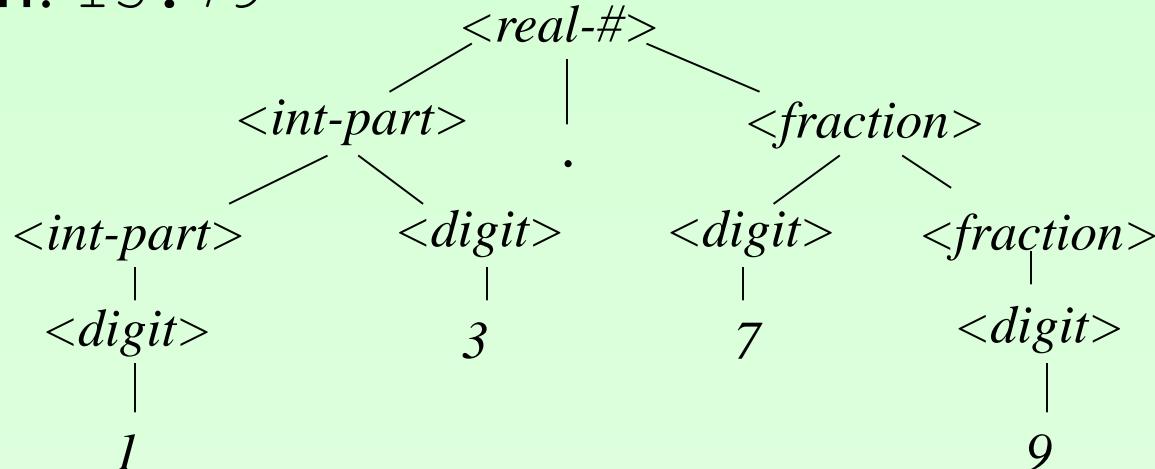
Usefulness of Grammar

Examples

- Description of (positive) real numbers:

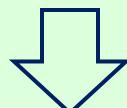
```
<real-#> ::= <int-part> . <fraction>
<int-part> ::= <digit> | <int-part> <digit>
<fraction> ::= <digit> | <digit> <fraction>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Token: 13.79



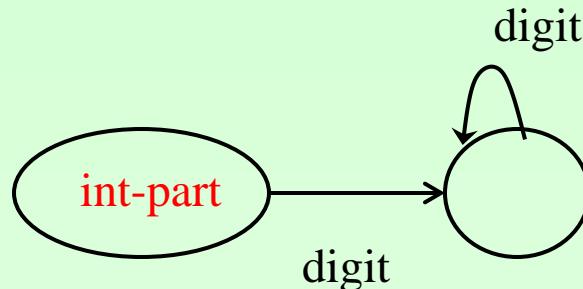
Regular Grammars

- A regular grammar (RF) is a subset of CFG:
 - A set of terminal symbols T (tokens or constants)
 - A set of non-terminal symbols N
 - One (non-terminal) start symbol σ
 - A set of grammar (rewriting) rules Ω of the form
 $\langle \text{nonterminal} \rangle ::= \langle \text{terminal} \rangle [\langle \text{nonterminal} \rangle]$
- Regular grammar can be expressed in terms of iterative construct of EBNF.

$$\begin{array}{lcl} \langle \text{int-part} \rangle & ::= & \langle \text{digit} \rangle \\ \langle \text{int-part} \rangle & ::= & \langle \text{digit} \rangle \langle \text{int-part} \rangle \end{array}$$

$$\langle \text{int-part} \rangle ::= \{ \langle \text{digit} \rangle \}^+$$

Regular Grammars

- Can be recognized by a finite state machine (and tail-recursive methods)

$$\langle \text{int-part} \rangle ::= \{ \langle \text{digit} \rangle \}^+$$


Implementation

- Two ways of recognizing regular grammars
 - Use tail-recursive methods
 - Use combinators for regular grammars

(Details are omitted in Lecture)

Parsers

A Generic Parser

- We can build generic parser.
- Each parser is expected to have the following type:

```
data Token = ...
```

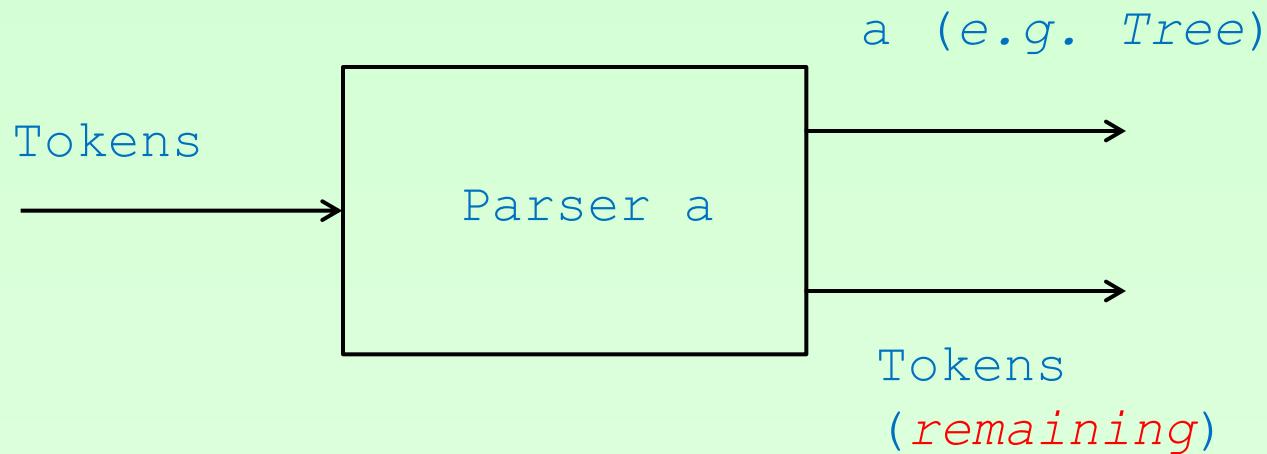
```
type Tokens = [Token]
```

```
type Parser a = Tokens -> (a, Tokens)
```

- It consumes some token so as to produce a parse tree object of type **a**.

A Generic Parser

```
type Parser a = Tokens -> (a, Tokens)
```



Keyword Parser

- Simple parsers would recognize keywords, identifiers and constants.

```
symbol :: String -> Parser ()  
  
symbol s (Key b:toks) =  
    if s == b then  
        ((), toks)  
    else  
        error ("Symbol "+s+" expected.")  
  
symbol s _ = error ("Symbol " + s + " expected.")
```

- Exception is thrown when expected keyword is not present.
- Why is unit returned?

Parser for Identifier

- A parser for identifier would return the string of the identifier.

```
ident :: Parser String
ident (ID a :: tokens) = (a, tokens)
ident _                  = error ("Identifier expected")
```

- Exception is thrown when identifier is not the next token.
- Note that the identifier token is consumed.

Parser for Constants

- A parser for constant would return the constant itself. In the case of integer, we expect the parser to have int type as its result.

```
numb :: Parser Int
numb (Num a:toks) = (a, toks)
numb _              = error ("Integer expected")
```

Exception is thrown when integer constant is not the next token.

Parser Combinators

- We need some ways to build bigger parsers from smaller parsers.
- We would like to have:
 - Sequential composition.
 - Alternative parsing.
 - Repetitive parsing
 - Optional parsing
 - A parser with a mapping transformer

Recall Notations

- $\langle x \rangle$ nonterminal x
- $\langle x \rangle ::= Body$ $\langle x \rangle$ is **defined by** $Body$
- $\langle x \rangle \mid \langle y \rangle$ either $\langle x \rangle$ **or** $\langle y \rangle$ (choice)
- $\langle x \rangle \langle y \rangle$ the sequence $\langle x \rangle$ followed by $\langle y \rangle$
- $\{ \langle x \rangle \}$ sequence of zero or more occurrences of $\langle x \rangle$
- $\{ \langle x \rangle \}^+$ sequence of ***one or more*** occurrences of $\langle x \rangle$
- $[\langle x \rangle]$ ***zero or one*** occurrence of $\langle x \rangle$

Monadic World

```
class Monad m where
    (">>=)      :: m a -> (a -> m b) -> m b
    return    :: a -> m a

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

class Functor f => Applicative f where
    pure     :: a -> f a
    (<*>)    :: f (a -> b) -> f a -> f b
    (<*>)    = liftA2 (\x->x)
    liftA2   :: (a -> b -> c) -> f a -> f b -> f c
    liftA2 f x = (<*>) (fmap f x)
```

Monadic World

```
class Applicative f => Alternative f where
    empty          :: f a
    (<|>)         :: f a -> f a -> f a

-- Example of instance

instance Alternative Maybe where
    empty          = Nothing
    Nothing <|> x = x
    Just x <|> _ = Just x
    -- Maybe can only hold up to one result,
    -- so we discard the second one.

instance Alternative [] where
    empty          = []
    (<|>)         = (++)
```

Do-Notation Sugar

Syntactic Sugar for bind operation:

conceptually:

```
m_expr ::= return expr  
         | m_expr >>= (\v -> m_expr)
```

m_expr >>= (\var -> **return** expr)

is same as

do

var <- m_expr

return expr

Sequential Parser Combinator

- Sequential parser applies one parser and then another.
- It can be implemented using:

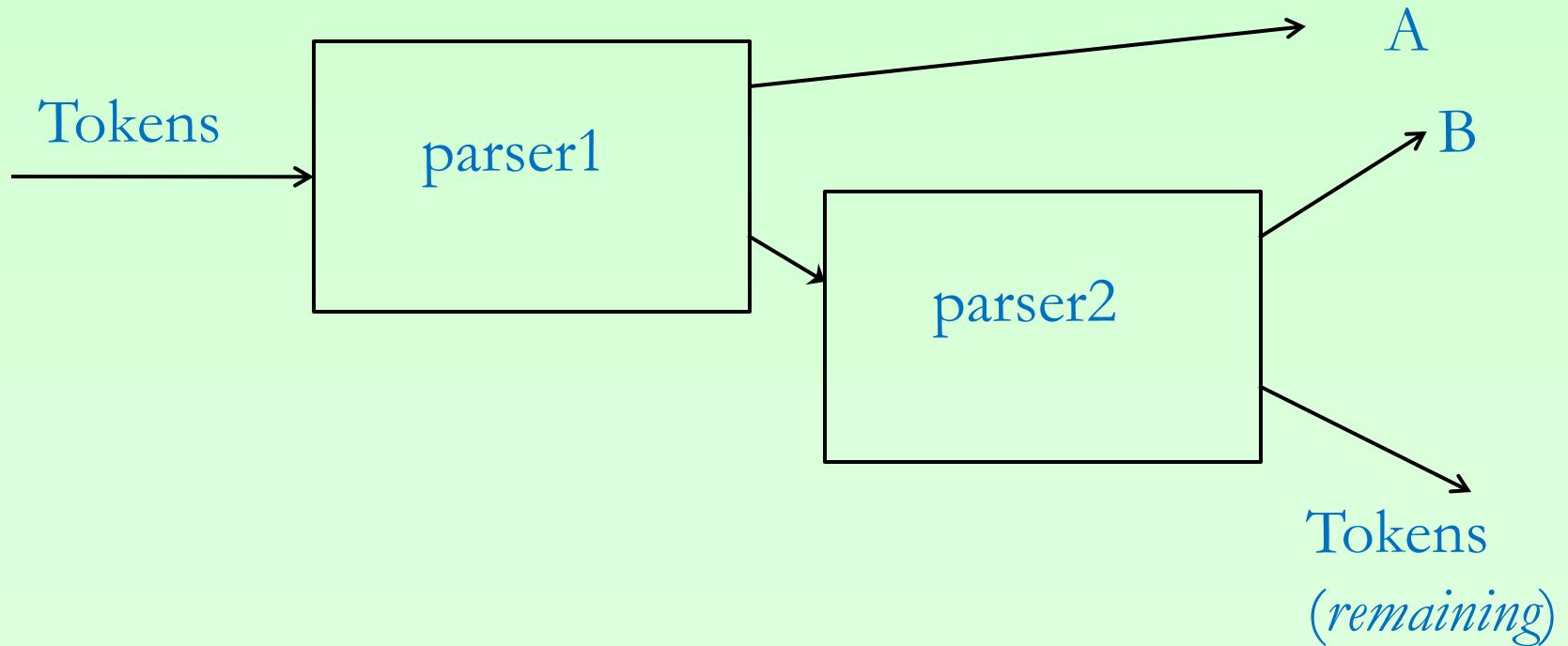
```
(~~) :: Monad SParsec => SParsec a -> SParsec b -> SParsec (a, b)
(~ ~) ph1 ph2 = do
    tup1 <- ph1
    tup2 <- ph2
    return (tup1, tup2)

hexPre = char '0' ~~ char 'x'
parse hexPre "(source-filename)" -- "0xAB0"
```

- Note that a tuple (a,b) is being returned.

A Generic Parser

parser1 $\sim\sim$ parser2



Alternative Parser Combinator

- Alternative combinator `<|>` allows a choice between two parsers.

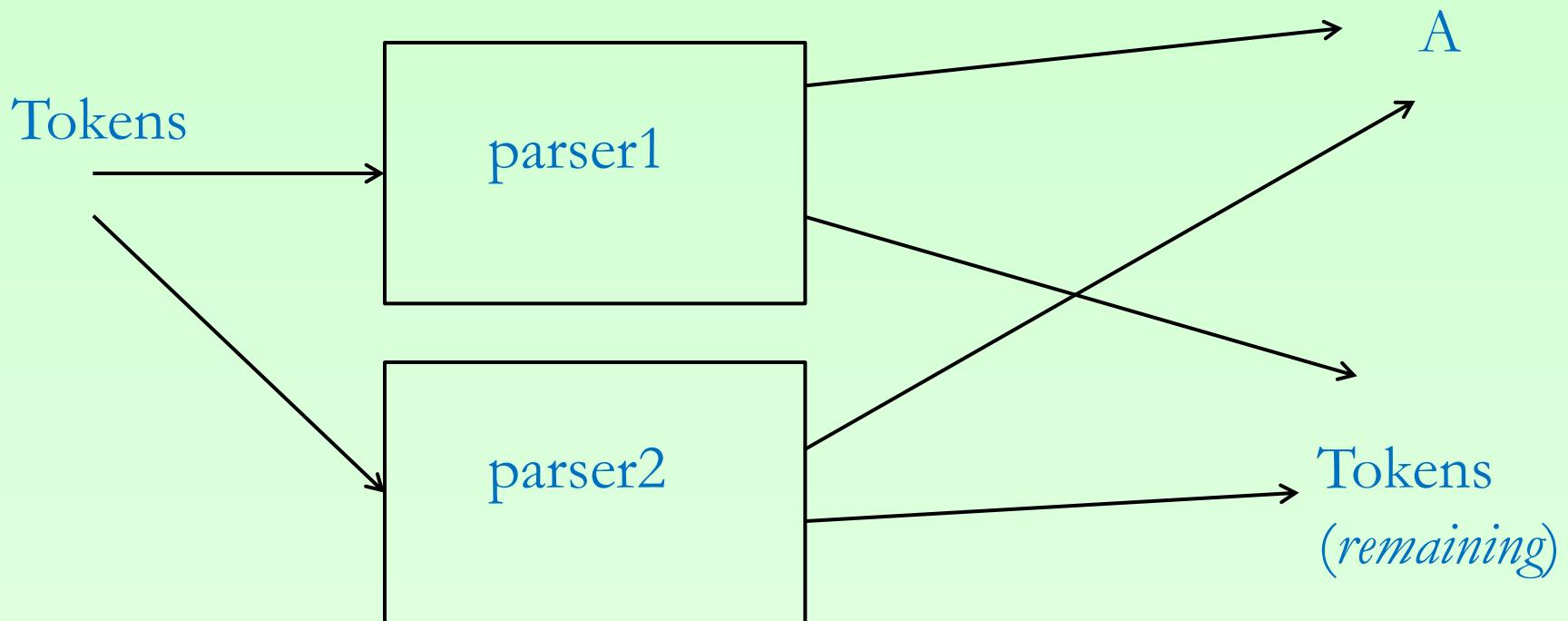
```
(<|>) :: SParsec a -> SParsec a -> SParsec a
```

```
parserA <|> parserB
```

- Recall that it tries the first alternative, and would only consider the second parser if the first parser fails and no input is consumed.

A Generic Parser

parser1 <|> parser2



Parser Mapping

- This is used to change the result of a parser by a map operation.
- It can be used to discard or transform outcomes:

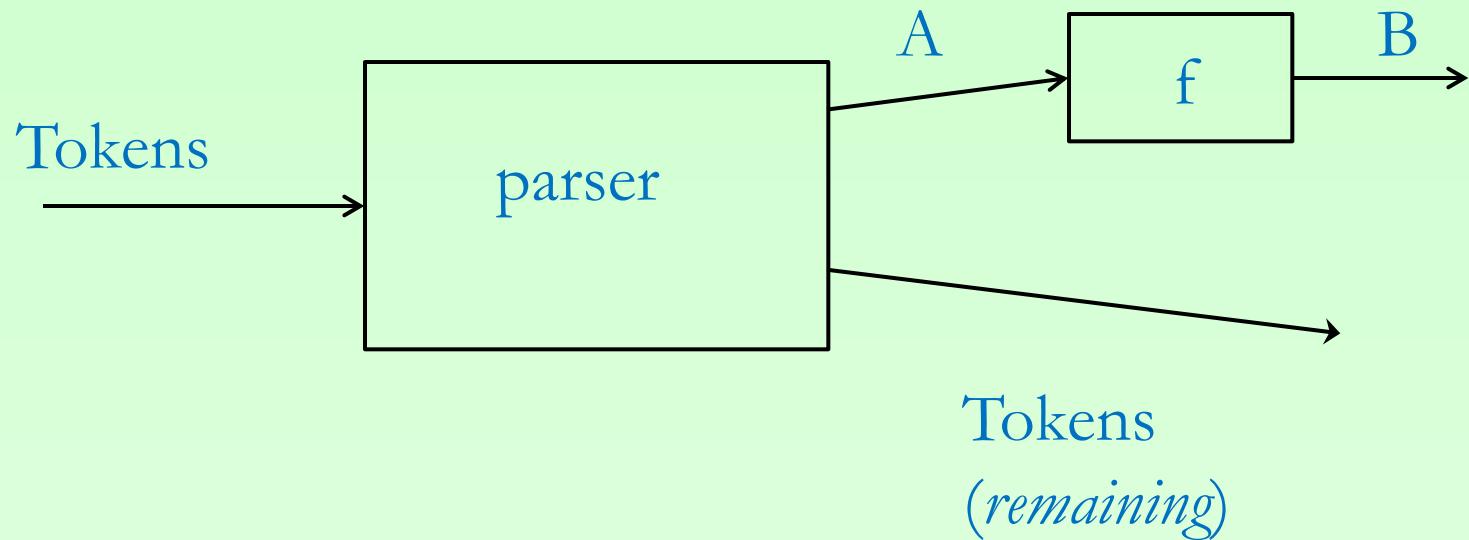
```
parsecMap :: (a -> b) -> SParsec a -> SParsec b
parsecMap f m = do
    x <- m
    return $ f x

parsecMap (read :: Int) (many1 digits)
```

- Function **f** changes the result of the parser.

A Generic Parser

parsecMap f parser



Repetitive Parsing

- We can repeat a parser zero or more times.
- It can be implemented using:

```
many   :: SParsec a -> SParsec [a]
many1  :: SParsec a -> SParsec [a]
many1 p = do x <- p
             xs <- many p
             return (x:xs)

many1 digits
```

- Note it repeats until the given parser fails.

Some Other Combinators

- We may have a parser optionally applied.

```
option      :: a -> SParsec a -> SParsec a
```

```
option x p = p <|> return x
```

```
choice     :: [SParsec a] -> SParsec a
```

```
choice ps = foldr (<|>) mzero ps
```

```
chainl     :: SParsec a -> SParsec (a -> a -> a) -> a -> SParsec a
```

```
chainl1    :: SParsec a -> SParsec (a -> a -> a) -> SParsec a
```

```
option 0 (many digits)
```

```
choice [char 'x', char 'y', char 'z']
```

Text.Parsec

- Combinators:

| | |
|-----------------------|---|
| p1 ~~ p2 | -- sequencing: must match p1 followed by p2 |
| p1 < > p2 | -- alternation: must match either p1 or p2, with preference given to p1 |
| p1 <?> st | -- may match p1 or show st as error message |
| try p1 | -- does not consume any input if fails |
| choice [p1,p2,...] | -- applies alternation sequentially |
| many p1 | -- repeating zero or more times |
| many1 p1 | -- repeating one or more times |
| skipMany p1 | -- like many but skips its result |
| between open p1 close | -- parses open, then p1, and at last close |
| parseMap f p1 | -- map function f to the parser's result |
| p1 *> p2 | -- like ~~ but ignore left member |
| p1 <*> p2 | -- like ~~ but ignore right member |

An Example :

Arithmetic Expression Parser

Arithmetic Grammar Rule

- Right-recursive to support right associativity of operators:

```
<fac> ::= <constant> | "(" <expr> ")"
<term> ::= <fac>          | <fac> ("*" | "/") <term>
<expr> ::= <term>         | <term> ("+" | "-") <expr>
```

- This runs but recursive case need to be considered where possible to cover larger expression.

Arithmetic Grammar Rule

- Left-recursive to support left associativity of operators:

```
<fac> ::= <id> | <constant> | "(" <expr> ")"
<term> ::= <fac> | <term> ("*" | "/") <fac>
<expr> ::= <term> | <expr> ("+" | "-") <term>
```

- Such left-recursion works well only if we had a non-deterministic parser since such parsers always terminate whenever there is a base case.

Arithmetic Grammar Rule

- Solution : Use repetition construct of EBNF to handle left associativity.

```
<fac>  ::= <constant> | "(" <expr> ")"
<term> ::= <fac> { ("*" | "/") <fac> }
<expr> ::= <term> { ("+" | "-") <term> }
```

Example Parser

```
module SParsec where

import Text.Parsec

data Expr = Const Int          | Plus Expr Expr
           | Minus Expr Expr | Mult Expr Expr
           | Div Expr Expr   deriving (Show)

eAdd x y = Plus x y
eSub x y = Minus x y
eMult x y = Mult x y
eDiv x y = Div x y
sToC s    = Const (read s)

type SParsec = Parsec String ()
```

Example Parser

expr :: TParsec Expr

expr = chainl1 term addop -- x+y-z+...

term :: TParsec Expr

term = chainl1 factor mulop -- x*y/z*...

factor :: TParsec Expr

factor = (parens expr) <|> constants -- 12 | (...)

parens :: TParsec Expr -> TParsec Expr

```
parens ex = do char '('  
              x <- ex  
              char ')'  
              return x
```

Example Parser

```
constants :: SParsec Expr
```

```
constants = parsecMap sToC $ many1 digit
```

```
digit :: SParsec Char
```

```
digit =
```

```
    char '0' <|> char '1' <|> char '2' <|>
```

```
    char '3' <|> char '4' <|> char '5' <|>
```

```
    char '6' <|> char '7' <|> char '8' <|> char '9'
```

```
mulop :: SParsec (Expr -> Expr -> Expr)
```

```
mulop = do { char '*'; return eMult }
```

```
    <|> do { char '/'; return eDiv }
```

```
addop :: SParsec (Expr -> Expr -> Expr)
```

```
addop = do { char '+'; return eAdd }
```

```
    <|> do { char '-'; return eSub }
```

Example Parser

```
calcE :: String -> Either ParseError Expr
calcE x = parse expr "" x

eval :: Either ParseError Expr -> Either ParseError Int
eval x =
    let eval' :: Expr -> Int
        eval' Const x = x
        eval' Add x y = eval x + eval y
        eval' Sub x y = eval x - eval y
        eval' Mult x y = eval x * eval y
        eval' Div x y = div (eval x) (eval y)
    in
        fmap eval' x
```

Tail-Recursive Implementation

Implementation Considerations

- We must identify reserved keywords and special symbols.
- We can support it using two list of reserved ids:

```
-- permitted symbols
symbols = ["=", "+", "-", "*", "/", "~", "(" , ")" ]  
  
-- reserved keywords in calculator
alphas = ["let", "in"]
```

Implementation for Lexical Analyser

- We must define the token type and a pretty printer:

```
data Token =  
    -- a lexical word  
    Key String  
  | Id String  
  | Num Int  
  -- exception for lexical analyser  
data Exception = LexErr String  
  
instance Show Token where  
    show Key s = "Key " ++ s  
    show Id s = "Id " ++ s  
    show Num n = "NumI " ++ (show n)
```

Implementation for Lexical Analyser

- Let us implement a scanner for alphanumeric identifier after a letter has been captured in parameter id.

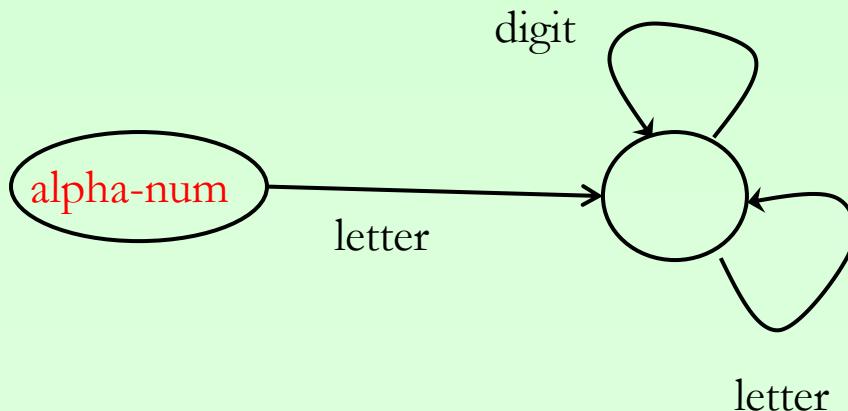
```
alphanum :: [Char] -> [Char] -> ([Char], [Char])
alphanum []      id = (reverse id, [])
alphanum (c:cs) id = if is_letter_or_digit c then
                      alphanum cs (c:id)
                  else
                      (reverse id, (c:cs))
```

- This implementation should be *tail-recursive* as tokens are expressible using regular grammar that can be recognized by finite automata.

Regular Grammars

Can be recognized by a finite state machine
(and tail-recursive methods)

$\langle \text{alphanum} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{letter} \rangle | \langle \text{digit} \rangle \}$



Identifying Alphanumeric Keyword

- Whenever we have identified an alphanumeric variable, we can convert it to a token by checking if it is a keyword or an identifier.

```
is_mem x = foldl false (\a z -> a || x=z)

tokenof :: [Char] -> Token
tokenof a = if is_mem a alphas then
            Key a
        else
            Id a
```

Implementation for Lexical Analyser

- Scanner for symbolic keyword would search for shortest possible match, as these symbolic keywords need *not* be separated by white spaces.

```
symbolic :: [Char] -> String -> (String, [Char])
symbolic []          sy = (sy, [])
symbolic (c:cs)      sy =
    if is_mem sy symbols then
        (sy,c:cs)
    else if not (is_mem c specials) then
        error ("Unrecognized lex symbol " ++ sy)
    else
        symbolic cs (sy ++ [c])
```

Regular Expression in Haskell

Regex

- Text.Regex.Posix gives basic functionality of regular expression in Haskell.
- Declare a date format with regular expression with three groups.

```
rgDate = "([1-2][0-9]{3})-([0-1]?[0-9])-([0-3]?[0-9])"
```

- ($=\sim$) is a polymorphic operator to match regular expression
- We can check if a string matches a regular expression

```
"2017-9-22" =~ rgDate :: Bool  
-- returns True
```

Regex

- We can get the matched sub-string

```
"2017-9-22 ..." =~ rgDate :: String  
-- returns "2017-9-22"
```

- We can retrieve all matches and the sub-expressions

```
"1.2017-9-22 2.1999-12-31" =~ rgDate :: [[String]]  
-- [[["2017-9-22", "2017", "9", "22"], ["1999-12-31", "1999", "12", "31"]]]
```

- We can also retrieve their offsets and lengths

```
"1.2017-9-22 2.1999-12-31" =~ rgDate :: [MatchArray]  
  
-- [array (0,3) [(0,(2,9)),(1,(2,4)),(2,(7,1)),(3,(9,2))],  
    array (0,3) [(0,(14,10)),(1,(14,4)),(2,(19,2)),(3,(22,2))]]
```

Regex

- Find all matches in a string

```
getAllTextMatches ("1.2017-9-22 2.1999-12-31" =~ rgDate)
  :: [String]
-- ["2017-9-22", "1999-12-31"]
```

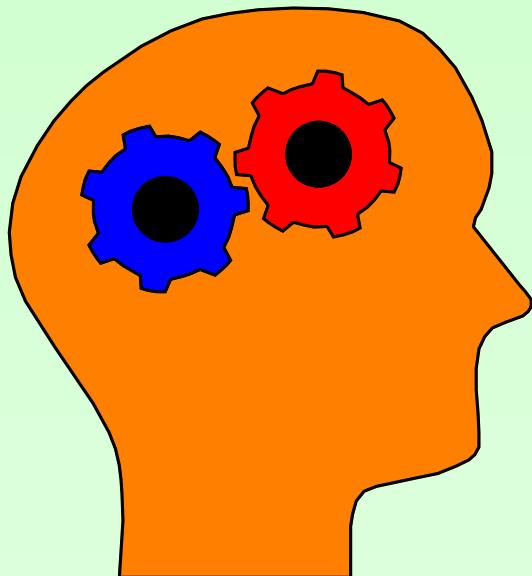
- Find string before matches, when matches, after matches, and all matched sub-expressions

```
"1.2017-9-22 2.1999-12-31" =~ rgDate
  :: (String, String, String, [String])
-- ("1.", "2017-9-22", " 2.1999-12-31", ["2017", "9", "22"])
```



CS2104: Programming Languages Concepts

Lecture 8-9 : Prolog & CLP



*“Logic, Relational and
Constraint Programming”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Prolog – Some Highlights

- Atoms, Variables & Terms
- Relations and Clauses
- Unification
- List Manipulation
- Arithmetic
- Backtracking, Cuts, Negation
- (Finite) Constraint Solving

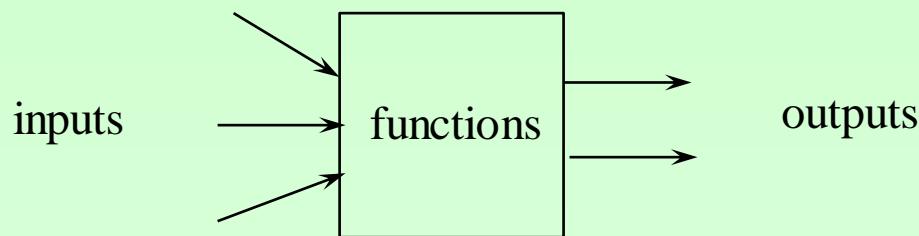
Reference --- An Introduction to Prolog Programming
<http://staff.science.uva.nl/~ulle/teaching/prolog/prolog.pdf>

Atoms, Terms and Variables

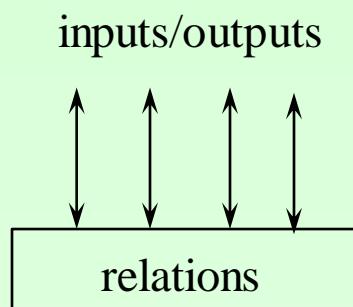
- Atoms are constants (starts with lower-case letter).
`cat, neil, john, 5, -1, mary, car`
- Variables start with *upper-case* letter or *underscore*
`X, Y, Y2, Result, _var, _1,`
- Terms are used to form tree-like data structures:
`node(node(dog,nil),leaf(cat)),
cons(2,nil), cons(cat,cons(1,nil))`
- Can mix terms with variables.
`node(X,Y), node(V,V), cons(2,T), cons(H,T)`
- Untyped language.

Relations vs Functions

- Prolog allows *relations* to be specified.
- This is facilitated by **unification** mechanism



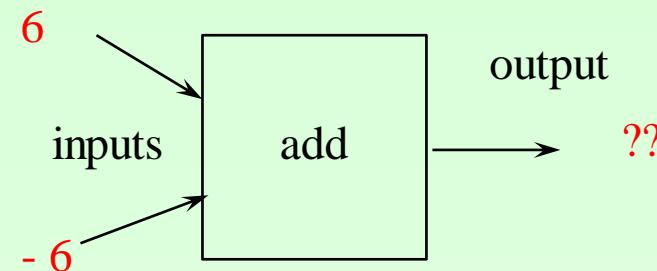
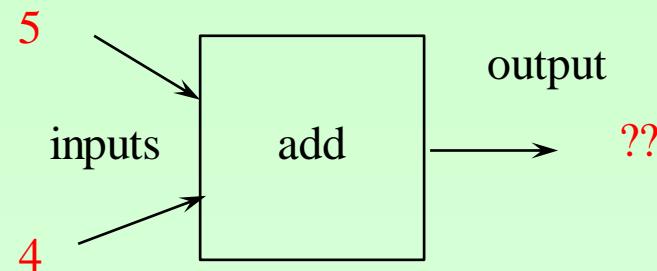
C, Java,
Haskell



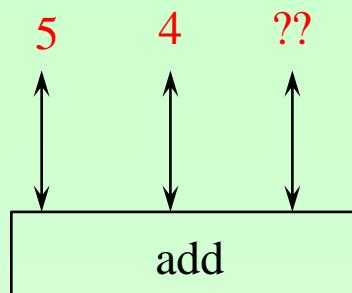
Prolog

Addition as a Function

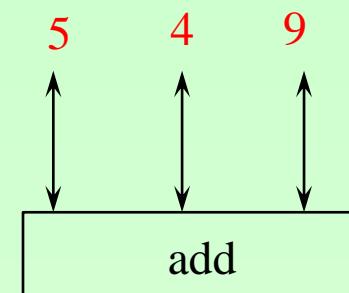
- Let us illustrate addition as a function.



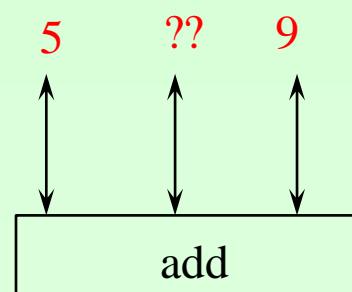
Addition as a Relation



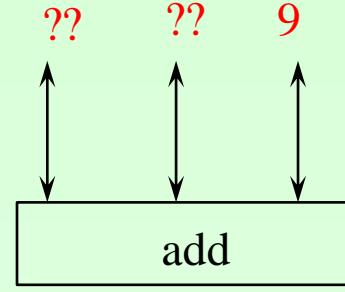
*adding
add(5, 4, R)*



*checking
add(5, 4, 9)*



*subtracting
add(5, Y, 9)*



*enumerating
add(X, Y, 9)*

Facts & Clauses

Relation via Facts

- We can provide facts as relations.

```
father(john, mary).  
father(john, tom).  
father(kevin, john).  
mother(eva, tom).  
mother(eva, mary).  
mother(cristina, john).  
male(john).  
male(kevin).  
male(tom).  
female(eva).  
female(cristina).  
female(mary).
```

Query on Facts

- Who is the father of mary?

father(x, mary).

- Who are child(ren) eva?

mother(eva, C).

- Who are daughter(s) of eva?

mother(eva, C), female(C).



denotes conjunction \wedge

Derived Facts via Horn Clauses

- Can construct Horn clause of the form:

```
pred(...) :- pred1(...) , pred2(...) , ... , predn(...) .
```

- Logical meaning:

$$\text{pred}_1(\dots) \wedge \text{pred}_2(\dots) \wedge \dots \wedge \text{pred}_n(\dots) \rightarrow \text{pred}(\dots)$$

Derived Facts via Horn Clauses

- Parent Relation:

```
parent(X,Y) :- father(X,Y).  
parent(X,Y) :- mother(X,Y).
```

- Another way to express disjunction:

```
parent(X,Y) :- father(X,Y); mother(X,Y).
```

↑
denotes disjunction \vee

Derived Facts via Horn Clauses

- Daughter:

```
daughter(X,Y) :- female(X), parent(Y,X).
```

- Sibling:

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\==Y.
```

- Grandparent:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

- Brother:

```
brother(X,Y) :- male(X), sibling(X,Y).
```

Recursive Horn Clauses

- Horn Clauses may be recursive
- How would you express the “ancestor” relation?

```
ancestor(X,Y) :- parent(X,Y).  
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

- Careful with left recursion : Infinite loop due to *depth-first* search procedure.

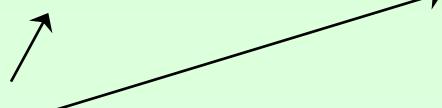
```
ancestor(X,Y) :- parent(X,Y).  
ancestor(X,Y) :- ancestor(Z,Y),parent(X,Z).
```

Unification

Unification by Example

- Unification is denoted by equality.
- Some examples:

a = X
a = b
n(a,X) = n(Y,b)
n(a,X) = n(X,b)
n(a,X) = n(X,a)
n(a,X) = n(X,p(a))
n(a,Y) = n(X,p(a))
n(Y,X) = n(X,p(a))



functor (or data constructor)

→ success X=a
→ fail
→ success X=b, Y=a
→ fail
→ success X=a
→ fail
→ success X=a, Y=p(a)
→ success X=p(a), Y=X

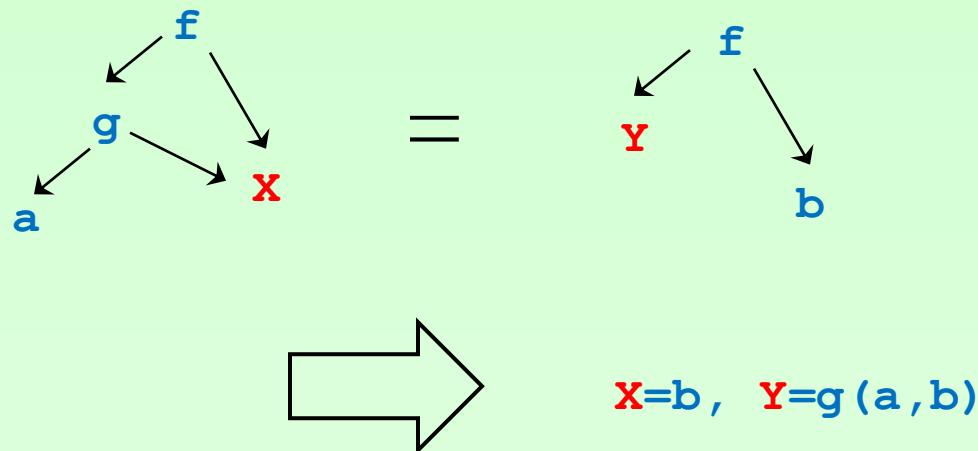
Essence of Unification

- Unification $t_1=t_2$ requests may contain variables.
- The system computes a *substitution* for the variables, so that two terms can be made equal.
- Once a variable become bound, it cannot be changed. This is essentially a *single-assignment* property.

Tree Representation of Unification

- Example:

?- $f(g(a, X), X) = f(Y, b)$



Unification Algorithm (no variables)

1. Initial unification request: $\Sigma_1 = \Pi_1, \Sigma_2 = \Pi_2, \dots$
2. If $\text{functor}(\Sigma_1) \neq \text{functor}(\Pi_1)$ or $\text{arity}(\Sigma_1) \neq \text{arity}(\Pi_1)$ then exit with failure.
3. If $\text{arity}(\Sigma_1) = 0$, remove $\Sigma_1 = \Pi_1$ from the unification request and go to last step.
4. Denote by $\Sigma_{11}, \Sigma_{12}, \dots, \Sigma_{1k}$, the arguments of Σ_1 and denote by $\Pi_{11}, \Pi_{12}, \dots, \Pi_{1k}$, the arguments of Π_1 .
5. Set the new unification request to:
$$\Sigma_{11} = \Pi_{11}, \Sigma_{12} = \Pi_{12}, \dots, \Sigma_{1k} = \Pi_{1k}, \Sigma_2 = \Pi_2, \dots$$
6. If current unification request is not empty, go to the first step. Otherwise, terminate with success

Unification Algorithm (with variables)

1. Initial unification request: $\Sigma_1 = \Pi_1, \Sigma_2 = \Pi_2, \dots$
If Σ_1 or Π_1 is a variable, add $\Sigma_1 = \Pi_1$ to the answer, and apply it as substitution to $\Sigma_2 = \Pi_2, \dots$ and go to last step
2. If $\text{functor}(\Sigma_1) \neq \text{functor}(\Pi_1)$ or $\text{arity}(\Sigma_1) \neq \text{arity}(\Pi_1)$ then exit with failure.
3. If $\text{arity}(\Sigma_1) = 0$, remove $\Sigma_1 = \Pi_1$ from the unification request and go to last step.
4. Denote by $\Sigma_{11}, \Sigma_{12}, \dots, \Sigma_{1k}$, the arguments of Σ_1 and denote by $\Pi_{11}, \Pi_{12}, \dots, \Pi_{1k}$, the arguments of Π_1 .
5. Set the new unification request to:
$$\Sigma_{11} = \Pi_{11}, \Sigma_{12} = \Pi_{12}, \dots, \Sigma_{1k} = \Pi_{1k}, \Sigma_2 = \Pi_2, \dots$$
6. If current unification request is not empty, go to the first step. Otherwise, terminate with success

Unification Algorithm Example

```
?- f(g(a,X),X) = f(Y, b)
```

Resolution

- *Resolution* : the process of answering a query.
- *Pattern-matching* is a special case of unification.
- Important concept : *variable renaming*.

All variables in a rule are replaced by completely new variables

- Example : `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)`
- 1st Renaming:
`ancestor(X1,Y1) :- parent(X1,Z1), ancestor(Z1,Y1)`
- 2nd Renaming:
`ancestor(X2,Y2) :- parent(X2,Z2), ancestor(Z2,Y2)`

Resolution Algorithm

1. Assume a query : A_1, A_2, \dots, A_n
2. Pick a matching rule from the program and *rename* its variables: $H :- B_1, B_2, \dots, B_k.$
3. New goal: $(H=A_1), B_1, B_2, \dots, B_k, A_2, \dots, A_n$
4. Variable bindings may be generated by the unification request $(H=A_1)$.
→ Add them to the answer, replace bound variables by its substitution over the entire query.
5. Continue from Step 1 until query is empty, and return the answer.

Resolution Demo

Video by Dr Razvan Voicu:

<http://www.youtube.com/watch?v=7-aKp-34iWE>

List

List Manipulation in Prolog

- List in Prolog is denoted by square bracket with its elements separated by comma:

[mary, [], n(A), [1,2,3], X]

- Prefix syntax also possible:

[t1,t2,t3] ≡ .(t1,.(t2,.(t3,[])))

- In order to break into head and tail, we can use either:

(i) .(H,T)

(ii) [H|T]

Append

- We can join two lists together by the following relation

```
append([], Y, Y).  
append([X|Xs], Y, [X|Rs]) :- append(Xs, Y, Rs).
```

- This is structurally similar to a functional definition with a base and a recursive scenario.

```
append([], Y)      =  Y  
append([X|Xs], Y) = [X|append(Xs, Y)]
```

- However, take note that the former is a relation, while the latter is a function.

Append

- In particular, relation can be executed in different ways.
- Joining two lists:

```
?- append([1,2,3],[4,5],Z).  
→ Z = [1,2,3,4,5]
```

- Computing the difference:

```
?- append([1,2,3],Y,[1,2,3,4,5]).  
→ Y = [4,5]
```

Append

- Splitting a List:

```
?- append(X,Y,[1,2]).  
→ X=[], Y=[1,2];  
    X=[1], Y=[2];  
    X=[1,2], Y=[].
```

- Prefix of a List:

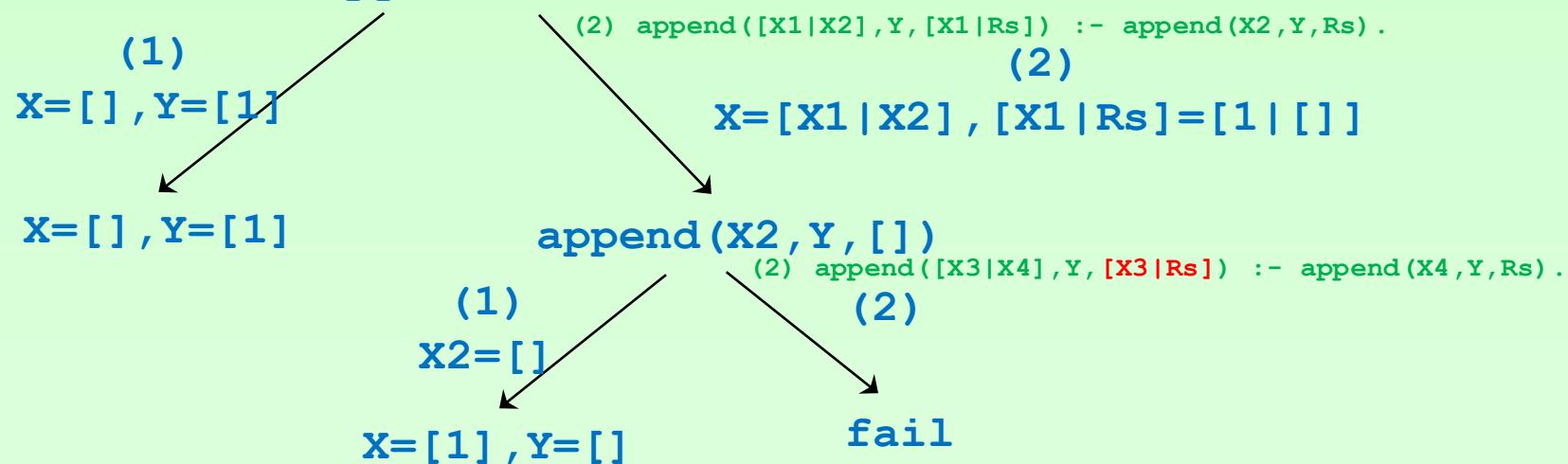
```
?- append(X,_,[1,2,3]).  
→ X=[]; X=[1]; X=[1,2]; X=[1,2,3].
```

Resolution/Search Tree

- We can represent the backtracking performed by resolution using a search tree.

```
(1) append([], Y, Y).  
(2) append([X|Xs], Y, [X|Rs]) :- append(Xs, Y, Rs).
```

- Consider: `append(X, Y, [1])`



Reverse

- We can reverse a list, as follows

```
reverse([], []).  
reverse([X|Xs], Y) :- reverse(Xs, Y2), append(Y2, [X], Y).
```

- This is similar to a functional definition

```
reverse([])      =  []  
reverse([X|Xs]) =  append(reverse(Xs), [X])
```

Reverse - Examples

- Reverse a List:

```
?- reverse([1,2,3],Y).  
→ Y=[3,2,1].
```

- Another way to reverse a list:

```
?- reverse(X,[1,2,3]).  
→ X=[3,2,1].
```

Reverse

- How about the following query? What does it compute?

```
?- reverse(X,X) .  
→      X=[] ;  
      X=[_V] ;  
      ...
```

Arithmetic

is-Operator

- Arithmetic expression is evaluated by the is-operator.
This is akin to functional evaluation.
- The 2nd argument of is-predicate must be a ground expression (without any variables) to allow the expression to be evaluated.

x is 3+4 → succeeded x=7

7 is 3+4 → succeeded

7 is X+4 → fail
(uninstantiated argument)

Comparator

- Relations `>`, `<`, `>=`, `=<`, `=\=`, `=:=` compares two arithmetic expressions that *must* be evaluated.
- The operator `=` is for term unification.

Factorial

- Computing factorial using a relation:

```
fact(0,1) .  
fact(N,R) :- N>0, M is N-1,  
            fact(M,R1), R is N*R1.
```

- Can compute:

```
?- fact(5,R) .  
      → R=120.  
?- fact(15,R) .  
      → R=1307674368000.
```

- But not : ?- fact(X,120) .
 **ERROR: >/2: Arguments are not
 sufficiently instantiated.**

Factorial

- Computing factorial using a relation:

```
fact(0,1) .  
fact(N,R) :- N>0, M is N-1,  
            fact(M,R1), R is N*R1.
```

- This above definition does not allow the first parameter **N** to be *unknown*.

Negation and Cut

Select

- List membership can be implemented as:

```
sel(X, [X|_]).  
sel(X, [_|T]) :- sel(X,T).
```

Note presence of non-linear pattern in LHS of Horn clause

- Functional version looks like:

| | |
|----------------------------|---------------------------------|
| <code>mem(X, [])</code> | <code>= false</code> |
| <code>mem(X, [Y T])</code> | <code>= if X=Y then true</code> |
| | <code>else mem(X,T)</code> |

- If you query:

```
?- sel(X, []).  
→ false
```

Fails immediately as both clauses are inapplicable.

Select

- List membership test:

```
?- sel(3, [1,3,5]).  
→ true
```

```
?- sel(6, [1,3,5]).  
→ false
```

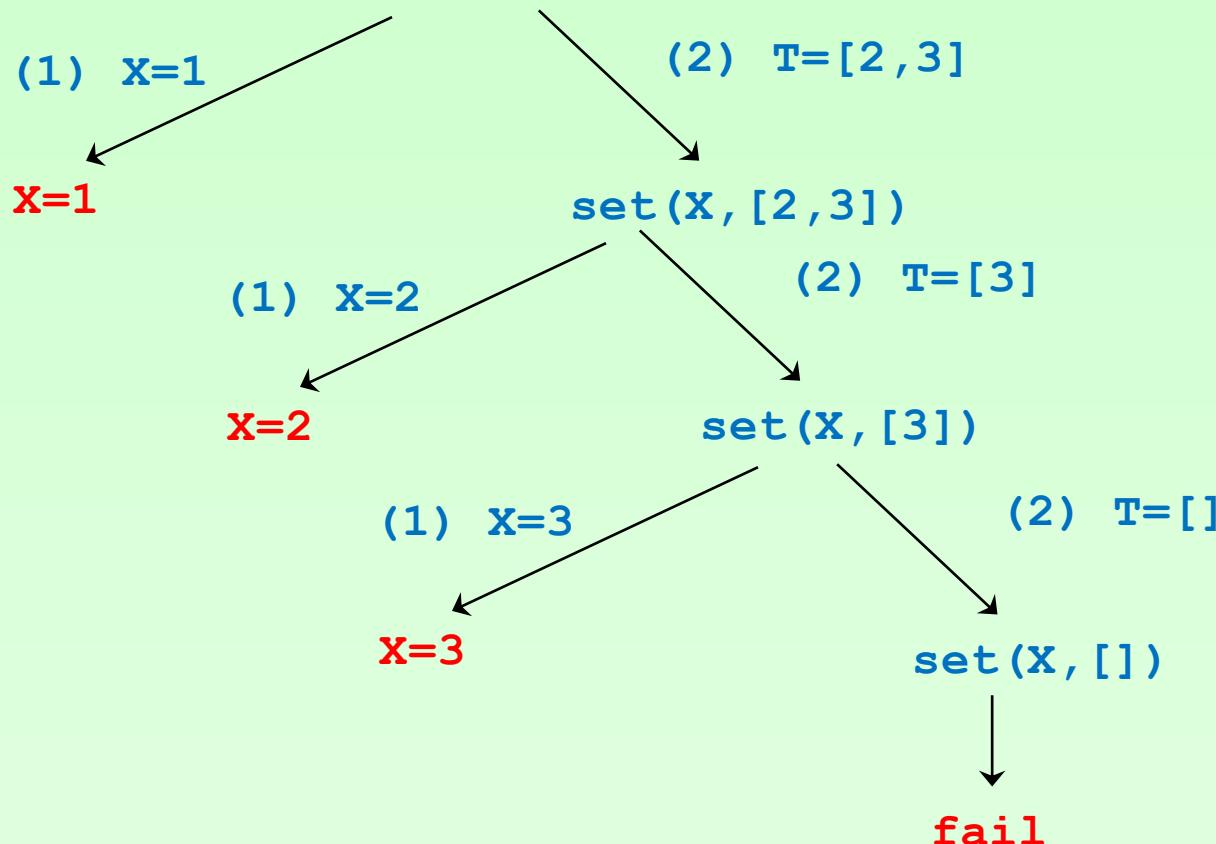
- Element generator:

```
?- sel(X, [1,3,5]).  
→ X=1 ; X=3 ; X=5.
```

Resolution/Search Tree

```
(1) sel(X, [X|_]).  
(2) sel(X, [_|T]) :- sel(X, T).
```

- Consider: `sel(X, [1,2,3])`



Negation as Failure

- Prolog is based on *closed* world assumption.
- Whatever can be proven is true.
- Whatever cannot be proven is *assumed* to be false.

Negation as Failure

- Consider.

```
father(john, mary).  
mother(eva, mary).  
father(john, tom).  
mother(eva, tom).  
father(kevin, john).  
mother(cristina, john).
```

- Query.

```
?- not(father(john,kerry)).  
→ true.
```

Negation as Failure

- Can use the clause.

```
male(X) :- not(female(X)).
```

- Need only specify facts on female.
- Above clause says if a person cannot be proven to be female, we shall assume the person is male.
- Negation as failure is sound only if the given clauses are complete.

Removing Duplicates

- To remove duplicate in a list.

```
remDupl([],[]).  
remDupl([H|T],R) :- sel(H,T), remDupl(T,R).  
remDupl([H|T],[H|R]) :- remDupl(T,R).
```

- Only partially correct:

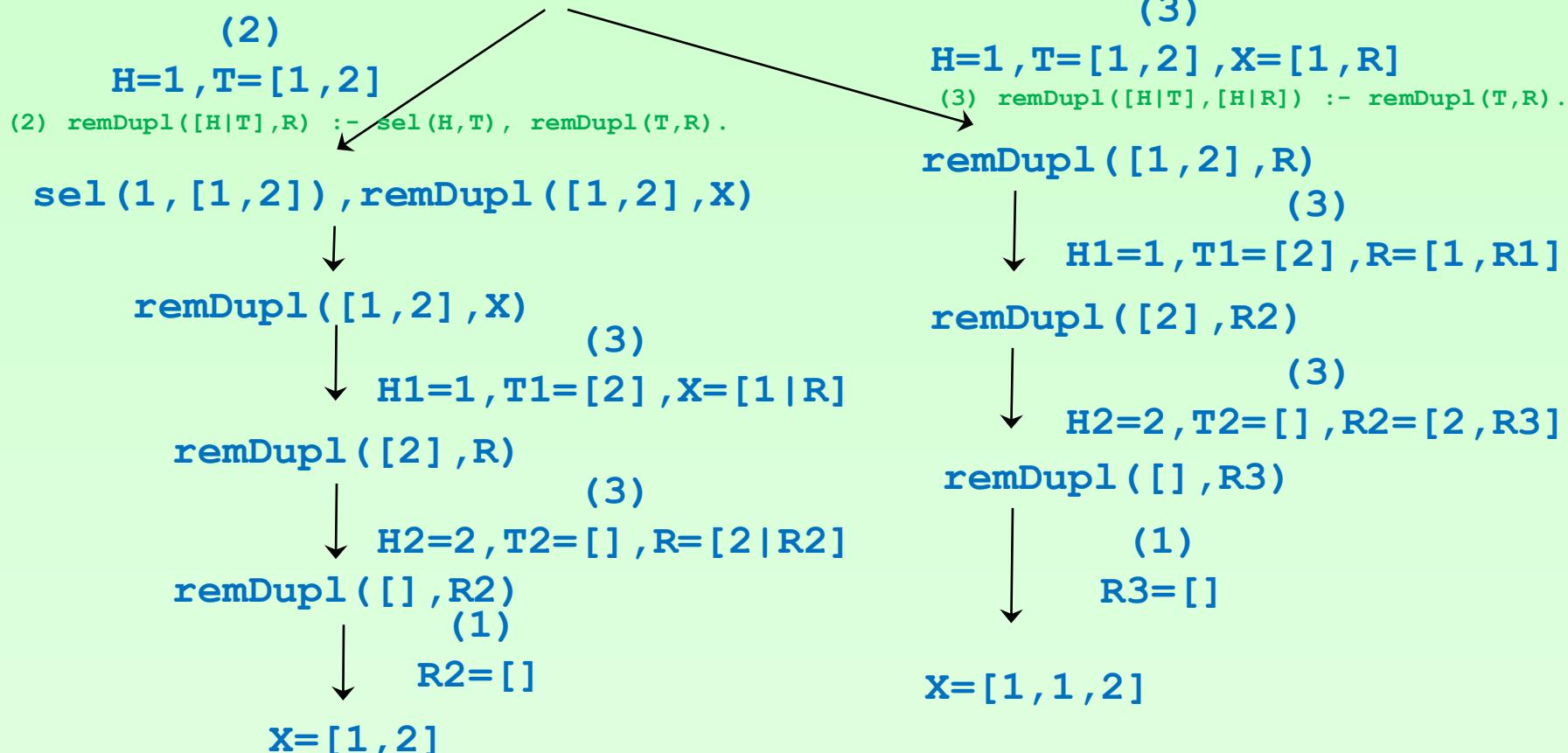
```
?- remDupl([1,1,2],R).  
→ R=[1,2];  
R=[1,1,2].
```

- First answer correct but not the second.

Resolution/Search Tree

```
(1) remDupl([],[]).  
(2) remDupl([H|T],R) :- sel(H,T), remDupl(T,R).  
(3) remDupl([H|T],[H|R]) :- remDupl(T,R).
```

- Consider: `remDupl([1,1,2],X)`



Using Negation

- Add a negation to 2nd clause.

```
remDup2( [], [] ).  
remDup2( [H|T] ,R) :- sel(H,T) , remDup2(T,R) .  
remDup2( [H|T] , [H|R]) :- not(sel(H,T)) , remDup2(T,R) .
```

- One correct solution:

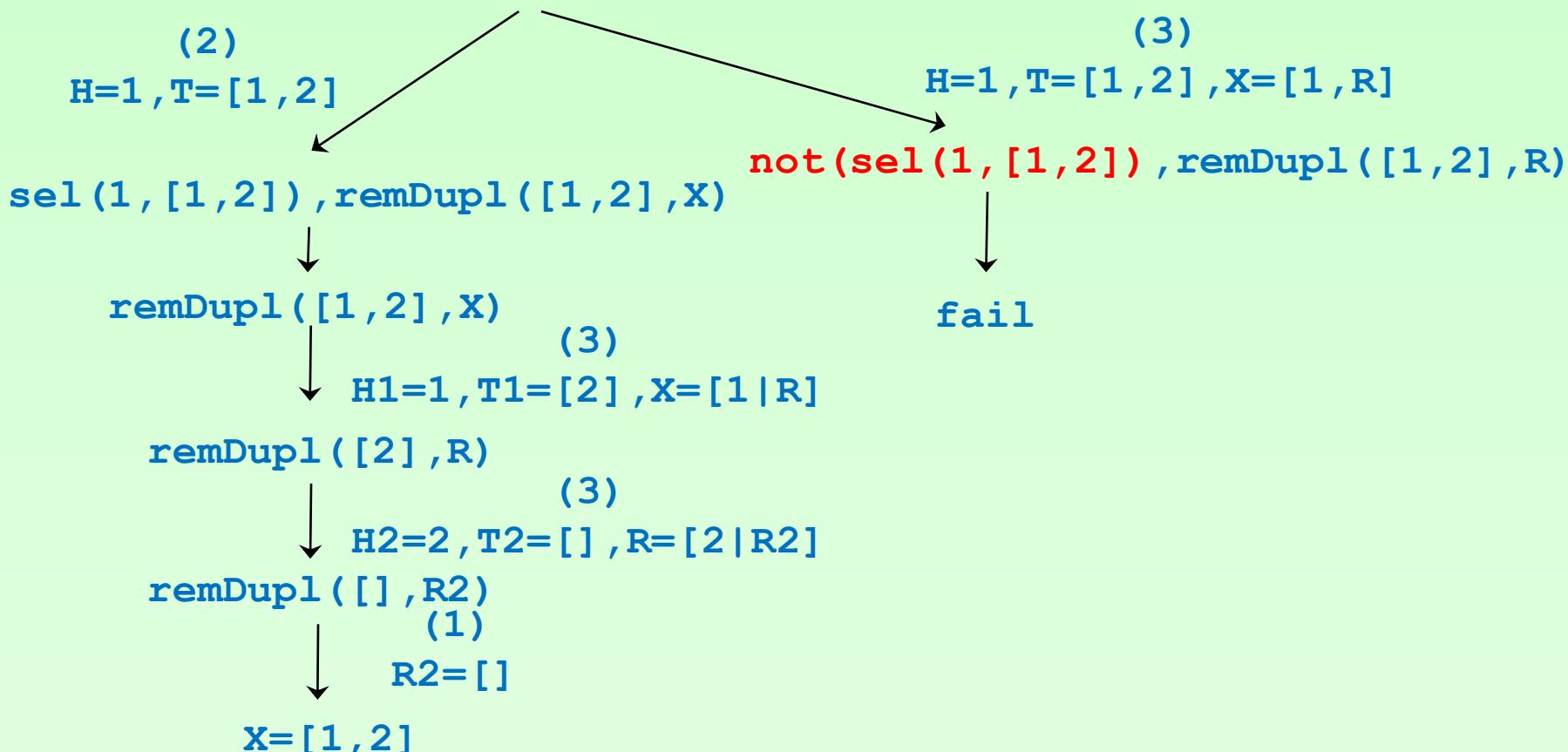
```
?- remDup2( [1,1,2] ,R) .  
→ R=[1,2] .
```

- This ensure that we try either 2nd or 3rd clause and not both.

Resolution/Search Tree

(1) `remDupl([],[]).`
(2) `remDupl([H|T],R) :- sel(H,T), remDupl(T,R).`
(3) `remDupl([H|T],[H|R]) :- not(sel(H,T)), remDupl(T,R).`

- Consider: `remDupl([1,1,2],X)`



Using Cut to Limit Backtracking

- To avoid backtracking, we can add a cut operator **!**.

```
remDup3([], []).  
remDup3([H|T], R) :- sel(H, T), !, remDup3(T, R).  
remDup3([H|T], [H|R]) :- remDup3(T, R).
```

- More efficient and one answer:

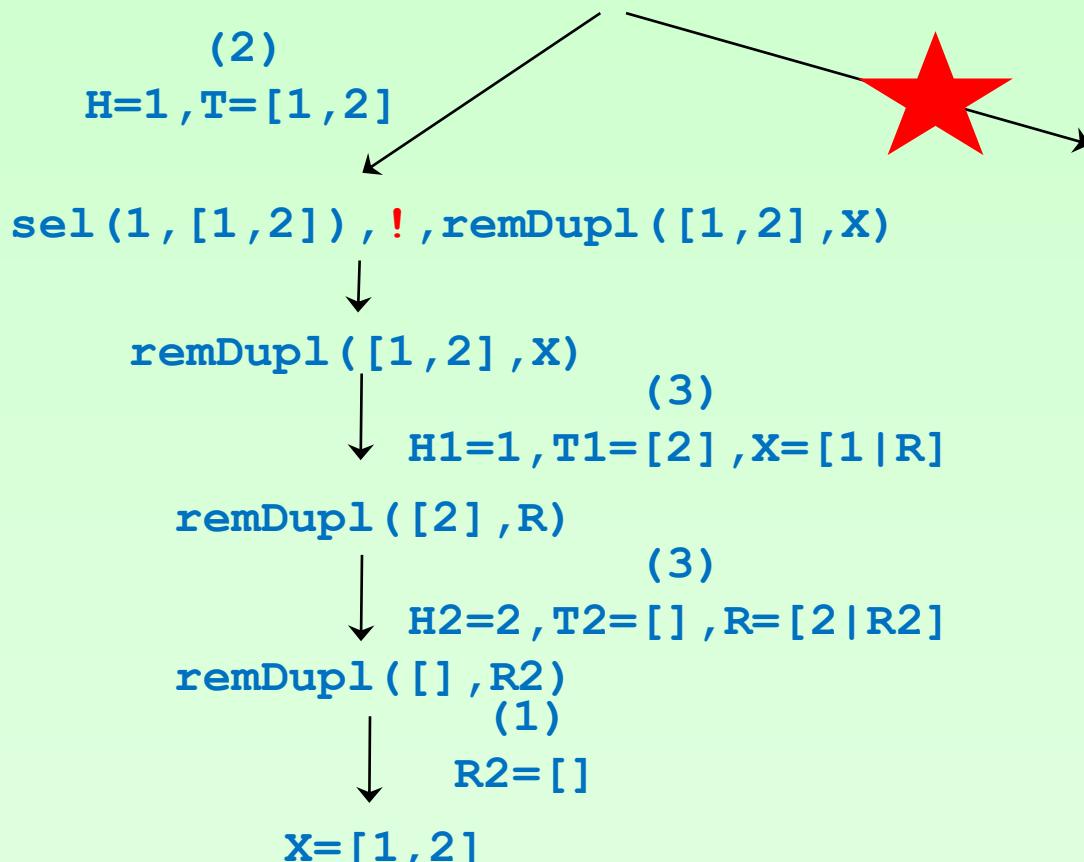
```
?- remDup3([1,1,2], R).  
→ R=[1,2].
```

- If **sel(H, T)** succeed, do not backtrack
- If **sel(H, T)** fails, backtrack to try 3rd clause.

Resolution/Search Tree

- (1) `remDupl([],[]).`
- (2) `remDupl([H|T],R) :- sel(H,T), !, remDupl(T,R).`
- (3) `remDupl([H|T],[H|R]) :- remDupl(T,R).`

- Consider: `remDupl([1,1,2],X)`



Impure Features

Impure Prolog

- **atom(X)** : succeeds if **X** is bound to an atom.
- **var(X)** : succeeds if **X** is a free variable.
- **integer(X)** : succeeds if **X** is bound to an integer.
- **write(X)** : output the binding of **X**.

Constraint Solving

Finite Constraint Solving

- Constraint solving based on finite domain.
- Essentially based on bounded arithmetic.
- Very powerful
 - Can solve puzzles.
 - Can be used for arithmetic relational.
 - Can be used for harder optimization problems.

Finite Domain Constraints

- Some basic constraints.

x #> 3.

→ x in 4..sup

x #\= 10.

→ x in inf..9\11..sup

3*x #= 9.

→ x=3

x*x #= 9.

→ x=3\/-3

Finite Domain Constraints

- More advanced constraints.

```
Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs),  
      X = 1, Y #\= 2.  
→ Vs = [1, 3, 2], X = 1, Y = 3, Z = 2.
```

```
4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.  
→ X = 3, Y = 6.
```

```
X #= Y #<==> B, X in 0..3, Y in 4..5.  
→ B = 0, X in 0..3, Y in 4..5.
```

Factorial with Finite Constraints

- Defining factorial more generally using finite constraint solving:

```
cfact(0,1).  
cfact(N,R) :- N#>0, M #= N-1,  
             R #= N*R1, cfact(M,R1).
```

- Note recursive call comes last, or solver may loop.
- Can now compute:

```
?- cfact(5,R).  
      → R=120.  
?- cfact(N,120).  
      → N=5.
```

Puzzle Solving

- Consider a coding system for alphabet that ensures the following:

$$\text{S E N D} + \text{M O R E} = \text{M O N E Y}$$

- A solution is to use:

$$S=9, E=5, N=6, D=7, M=1, O=0, Y=2$$

since we can show.

$$9 5 6 7 + 1 0 8 5 = 1 0 6 5 2$$

Puzzle Solving

- We can model this as a finite constraint problem using:

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-  
    Vars = [S,E,N,D,M,O,R,Y],  
    Vars ins 0..9,  
    all_different(Vars),  
    S*1000 + E*100 + N*10 + D +  
    M*1000 + O*100 + R*10 + E #=  
    M*10000 + O*1000 + N*100 + E*10 + Y,  
    M #\= 0, S #\= 0.
```

Puzzle Solving

- Querying with `puzzle(As+Bs=Cs)` gives partially solved answer:

```
As = [9, _G10134, _G10137, _G10140],  
Bs = [1, 0, _G10155, _G10134],  
Cs = [1, 0, _G10137, _G10134, _G10179],  
_G10134 in 4..7,  
all_different([9, _G10134, _G10137, _G10140, 1, 0,  
_G10155, _G10179]),  
1000*9+91*_G10134+ -90*_G10137+_G10140+ -9000*1+ -  
900*0+10*_G10155+ -1* _G10179#=0,  
_G10137 in 5..8,  
_G10140 in 2..8,  
_G10155 in 2..8,  
_G10179 in 2..8.
```

Puzzle Solving

- Using `label(As)` that performs an *enumeration* gives unique solution!

```
puzzle(As+Bs=Cs) ,label(As) .  
As = [9, 5, 6, 7] ,  
Bs = [1, 0, 8, 5] ,  
Cs = [1, 0, 6, 5, 2] ;
```

Logic Puzzle

There are 5 houses, each of a different color, and inhabited by a person from a different country who has a different pet, drink, and make of a car.

- (a) The English woman lives in the red house.
- (b) The Spaniard owns the dog.
- (c) Coffee is drunk in the green house.
- (d) The Ukrainian drinks tea.
- (e) The green house is immediately to the right of the ivory house.
- (f) The BMW driver owns snails.
- (g) The owner of the yellow house drives a Toyota.
- (h) Milk is drunk in the middle house.
- (i) The Norwegian lives in the first house of the left.
- (j) The person who drives the Ford lives in the house next to the owner of the fox.
- (k) The Toyota driver lives in the house next to the house where the horse is kept.
- (l) The Honda owner drinks orange juice.
- (m) The Japanese drives a Datsun.
- (n) The Norwegian lives next to the blue house.

Logic Puzzle

There are 5 houses, each of a different color, and inhabited by a person from a different country who has a different pet, drink, and make of a car.

People = [English, Spaniard, Ukrainian, Norwegian, Japanese],
Colors = [Red, Green, Ivory, Yellow, Blue],
Drinks = [Tea, Milk, Orange, Coffee, Water],
Pets = [Dog, Snails, Fox, Horse, Zebra],
Cars = [BMW, Toyota, Ford, Datsun, Honda],
Houses = [House1=1, House2=2, House3=3, House4=4, House5=5]

L = [People,Colors,Drinks,Pets,Cars],

Logic Puzzle

Adding constraint on all_different

appendall(L, All),

All ins 1..5,

all_different(People), all_different(Colors),

all_different(Drinks), all_different(Pets),

all_different(Cars),

Logic Puzzle

(a) The English woman lives in the red house.

English = Red

(b) The Spaniard owns the dog.

Spaniard = Dog

(c) Coffee is drunk in the green house.

(d) The Ukrainian drinks tea.

(e) The green house is immediately to the right of the ivory house.

Green #= Ivory+1

(f) The BMW driver owns snails.

(g) The owner of the yellow house drives a Toyota.

(h) Milk is drunk in the middle house.

Logic Puzzle

- (i) The Norwegian lives in the first house of the left.
- (j) The person who drives the Ford lives in the house next to the owner of the fox.
 $\text{abs}(\text{Ford}-\text{Fox}) \#= 1$
- (k) The Toyota driver lives in the house next to the house where the horse is kept.
- (l) The Honda owner drinks orange juice.
- (m) The Japanese drives a Datsun.
- (n) The Norwegian lives next to the blue house.

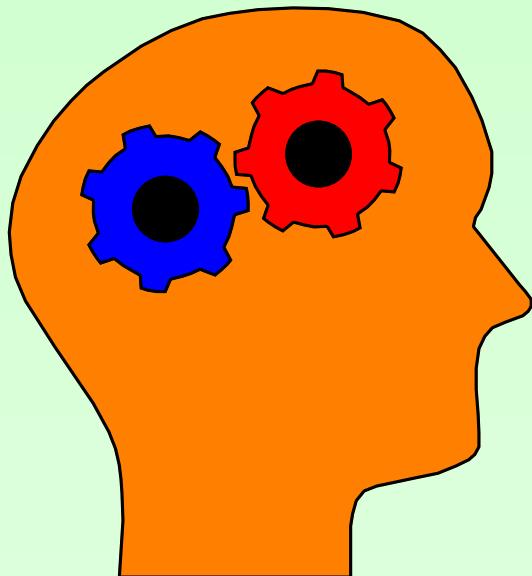
Summary

- Logic programming and particularly with constraint-solving capability is an extremely powerful problem-solving mechanism.
- Two current weaknesses
 - Untyped language setting
 - Lack of higher-order functions
- Nevertheless, special language features allows harder problems to be more easily modelled.



CS2104: Programming Languages Concepts

Lecture 10 : OCaml Basics and Imperative Programming



*“Basics of OCaml and
Imperative Programming”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Topics

- Basics of OCaml
- Mutable Reference & Records
- Input/Output Functions
- Iterators and Loops
- Arrays, String, Hash-Tables Modules
- Memoization

OCaml vs Haskell

| OCaml | Haskell |
|---------------------|----------------------|
| strict (by default) | lazy (by default) |
| impure | pure |
| let & let rec | let (rec by default) |
| Impure IO | Monadic I/O |
| Modules/Functors | Type classes |
| OOP | Layout Rule |
| | Comprehension |

Non-Recursive Let

- In OCaml, `let` binding is non-recursive.

```
let x = 1 in      // 1
let x = x-3 in   // 1-3
```

- For (mutual) recursive, use `let rec` instead:

```
let rec f x = ...g x...
    and g x = ...f x...g x...
in ...
```

- Non-recursive let supports shadowing and allows reuse of names.

Post-fix Type Application

- Type variables are written as ``a` in OCaml

- In Haskell, prefix type application used

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

- In OCaml, post-fix type application used

```
map :: (`a->`b) -> `a list -> `b list
```

Strict Evaluation

- let binding is strictly evaluated.

```
let x = e1 in      // e1 is strictly evaluated  
let y = e2 in      // e2 is strictly evaluated  
...
```

- Arguments are strictly evaluated but the order of evaluation is not specified

```
f e1 e2 // {e1,e2} are strictly evaluated
```

- Use `let` if argument order is important.

```
let v1 = e1 in // e1 is evaluated  
let v2 = e2 in // e2 is evaluated  
f v1 v2       // call is invoked
```

Lazy Module

- Laziness in OCaml is captured through type

`t Lazy.t`

- Deferred computation is built using:

`let (v:'a Lazy.t) = lazy (e:'a)`

- It is strictly evaluated through:

`force : 'a Lazy.t -> 'a
... force v ...`

- Can check if already evaluated using:

`is_val : 'a Lazy.t -> bool`

Pure versus Imperative

Pure versus Imperative

- Thus far, used mostly pure functions (without side-effects).
- Benefits of Pure Functions
 - Easier to reason/debug
 - Less error prone
 - Easily composable

When is Imperative Needed?

- Imperative feature also important
 - To model the real-world
 - For some high performance considerations (e.g. hash tables & mutable graphs)
- Compromise:
 - Use pure functions *where possible* (by default), and imperative features *where necessary*.

Mutable Reference and Fields

Mutable Reference

- Reference type supports pointer to a memory location that can be updated.

```
(* creates a mutable reference ptr *)
let ptr = ref 10;;

(* prints content of ptr *)
print_endline (string_of_int !ptr);;

(* update the content of ptr *)
ptr := !ptr + 1;; 

print_endline (string_of_int !ptr);;
```

Creating Mutable Reference

- Command `ref` has type `'a → 'a ref`.

```
(* creates a mutable reference ptr *)
let ptr = ref 10;;
let ptr2 = ref "hello";;
```

- In above example:
 - `ptr` has `(int ref)` type.
 - `ptr2` has `(string ref)` type

Dereferencing Mutable Locations

- Command `!` has type `'a ref → 'a.`

```
(* returns content of a mutable reference *)
let v:int = !ptr;;
let s:string = !ptr2;;
```

- In follow-up example:
 - `v` has `int` type.
 - `s` has `string` type

Updating Mutable Reference

- Command `:=` has type `'a ref → 'a → unit`.

```
(* updating a mutable reference *)
let () = ptr := !ptr + 1;;
let () = ptr2 := !ptr2 ^ " there!";;
```

- In follow-up example:
 - `ptr` content will change to `11`.
 - `ptr2` will change to `"hello there!"`
- Assignment returns unit (or void) type.

Mutable Fields of Record

- Records are immutable by default, but we can selectively provide fields that are mutable.
- Example:

```
(* declaring a record type *)
type ('a, 'b) pair =
  { mutable first : 'a;
    second : 'b
  } ;;

(* constructing a record value *)
let p1 = {first = 1; second = "cs2104"};;
```

Retrieve Fields of Record

- We can retrieve mutable and immutable fields in the same way, namely **record.field**.

```
print_endline ("First: "^(string_of_int p1.first));  
print_endline ("Second: "^(p1.second));
```

Updating Mutable Fields Only

- Only mutable fields can be updated.

```
(* update to mutable field *)
p1.first <- p1.first + 1;;
```

```
(* compile time error! *)
p1.second <- p1.second^" hello";;
```

→ Error: The record field label second is not mutable

Implementation of Ref Type

- How are reference types related to mutable fields of records?
- Each ref type is implemented as a record with exactly one mutable field!
- Thus, mutable fields are the fundamental construct in OCaml.

Implementation of Ref Type

```
(* type declaration *)
type 'a ref = { mutable contents : 'a };;
```



```
(* construction *)
let ref v = { contents = v };;
```



```
(* retrieval *)
let (!) r = r.contents;;
```



```
(* update *)
let (:=) r v = r.contents <- v;
```

Weak Polymorphism

- Types of mutable fields must not be polymorphic. Following is invalid:

```
let (p4: ('a list) ref) = ref [];;
```

```
p4 := 1::!p4;
```

```
p4 := "hello"::!p4;; (* error *)
```

- Mutable reference must have a single type, so that update and retrieval remains consistent.

Weak Polymorphism

- Immutable values can be polymorphic:

```
let id x = x;;
(* id: forall 'a. 'a -> 'a *)
```

- Mutable reference can only be monomorphic.

```
let p_id = ref id;;
(* p_id : ('_a -> '_a) ref *)
```

weak polymorphism
that must be bound
to a monotype

→ Error: The type of this expression, ('_a -> '_a) ref,
contains type variables that cannot be generalized

Weak Polymorphism

- Weakly polymorphic type must be instantiated to a monotype within the same module.

```
let p_id = ref id;;
(* p_id : ('_a -> '_a) ref *)
```

```
let v = !p_id 3;;
(* forces '_a be instantiated to int *)
```

```
let s = !p_id "hello";;
(* contradictory use cause type error *)
```

An Example

- Remembering first value.

```
let memo =  
  let cache = ref None in  
  (fun x ->  
    match !cache with  
    | Some y -> y  
    | None -> cache := Some x; x);;  
(* val memo : '_a -> '_a = <fun> *)  
  
memo 3;; (* --> 3 *)  
memo 4;; (* --> 3 *)
```

Wrongly Placed

- Cache is now local to each call !
where a new cache is created for each call.

```
let memo =
  (fun x ->
    let cache = ref None in
    match !cache with
    | Some y -> y
    | None -> cache := Some x; x);;
```

```
memo 3;; (* --> 3 *)
```

```
memo 4;; (* --> 4 *)
```

Input/Output

Input/Output

- Apart from data mutation, I/O is the other major source of side-effects.
- I/O involves interactions with the real world.
- Many I/O libraries including Asynchronous I/O library.

Buffered I/O Library

- This allows buffering to optimize interaction with the Unix I/O library.
- Two types of channels:
 - `in_channel` (for reading)
 - `out_channel` (for writing)

Common I/O Channels

stdin (* standard input *)

stdout (* standard output*)

stderr (* standard error *)

Interact with Terminal

```
let test () =
    output_string stdout
        "What is your name?";
flush stdout;
let ans = input_line stdin in
    output_string stdout
        ("Hello " ^ ans ^ "\n");;
```

Output to a File

```
let file = open_out "test.out";;
(* creates an out_channel file *)

output_string file "Hello There!";;
(* writes to file *)

close_out file;;
(* closes the file *)
```

Formatted Printing

- Formatted printing takes a format string to determine how printing is to be done.

```
let pr = Printf.printf
  ("%i is an integer, %F is a float"^^
  "%\"%s\\"" is a string\n");;
```

```
pr 3 4.5 "five";;
```

```
→ 3 is an integer, 4.5 is a float, "five" is a string
```

Formatted Printing

- OCaml uses a type-safe solution:

```
let fmt = "%i is an integer, %F is a  
float"^^"\%s\" is a string\n";;  
  
(int -> float -> string -> 'c, 'b, 'c) fmt  
  
let pr = Printf.printf fmt;;  
  
pr : int -> float -> string -> ()  
  
pr 3 4.5 "five";;
```

Loops and Iterators

Loop Iterators

- OCaml provides for/while loops to make it easier for imperative programming.

```
for i = 0 to 3 do
    printf "i = %d\n" i done;;
→
i = 0
i = 1
i = 2
i = 3
- : unit = ()
```

Note that i itself is local and immutable

Loop Iterators

- A count-down for-loop.

```
for i = 3 downto 0 do
    printf "i = %d\n" i done;;
→
i = 3
i = 2
i = 1
i = 0
- : unit = ()
```

Loop Iterators

- A while-loop.

```
let i = ref 3;;
while (!i>=0) do
  printf "i = %d\n" !i;
  i := !i-1
done;;
→      i = 3
        i = 2
        i = 1
        i = 0
- : unit = ()
```

Loop Iterators

- Implementation of for_loop using higher-order

```
let for_loop init final stmt =
  let rec aux i =
    if i<=final
    then (stmt i; aux (i+1))
  in aux init
```

- Exercise : write recursive codes for while and for_downto loops.

List Iterator

- We also have iterators over data structures.

`List.iter : ('a -> unit) -> 'a list -> unit`

`List.iter f [a1; ...; an]`

applies function `f` in turn to `[a1; ...; an]`.

It is equivalent to

`begin f a1; f a2; ...; f an; () end.`

Sequences

Sequences

- Sequences of the form: `e1;e2;...;en` are executed for their effects.
- It is equivalent to:

```
let _ = e1 in  
let _ = e2 in  
...  
en
```

Sequences

- Safer to use :

```
let () = e1 in  
let () = e2 in  
...  
en
```

- All except **en** are of the unit type. Why?

Evaluation Order

- Evaluation-order of arguments are implementation-dependent!
- What is the outcome of below?

```
let eval x =
    printf "Elem %d\n" x; x ;;
[eval 1; eval 2]
→
Elem 2
Elem 1
- : int list = [1; 2]
```

Evaluation Order

- Similarly:

```
let eval x =
    printf "Elem %d\n" x; x ;;
(eval 1, eval 2)
```



Elem 2

Elem 1

- : int * int = (1, 2)

Mutable Modules

Mutable Modules

- Quite a number of modules have mutable data structures.
- Examples are : Arrays, Strings, Hash Tables.
- They could be used to support both imperative and functional-style programming.

Array

- Key operators.

val make : int -> 'a -> 'a array

(* make n v returns a mutable array of size n, with initial value v *)

val length : 'a array -> int

(* returns size of array *)

val get : 'a array -> int -> 'a

(* get a n returns n-th elem of array a *)

val set: 'a array -> int -> 'a -> unit

(* set a n v updates n-th elem of array a with new value v *)

Array

- Short-hands.

```
val get : 'a array -> int -> 'a  
(* Array.get a n = a(n) *)
```

```
val set: 'a array -> int -> 'a -> unit  
(* Array.set a n v = a(n)<-v *)
```

Array

- Useful Higher-Order Functions.

```
val map : (int -> 'a -> 'b) ->  
  'a array -> 'b array
```

```
val fold_left : ('a -> 'b -> 'a) ->  
  'a -> 'b array -> 'a
```

```
val fold_right : ('b -> 'a -> 'a) ->  
  'b array -> 'a -> 'a
```

String

- We have used only string in a functional way.
- String are not only compact (8 characters per memory word), it also supports mutation:

```
val set: string -> int -> char -> unit
(* set s n c updates n-th elem of
   string s with value char c
   short-hand s.[n]<-c
*)
```

Hash Table (Hashtbl)

- Hash tables implement generic dictionary.

```
module Hashtbl
```

```
type ('a, 'b) t
```

optional labeled
parameter

```
val create : ?random:bool -> int  
           -> ('a, 'b) t
```

(* Hashtbl.create n will create a new hash table of initial size n, while ~random:true allows a randomized seed to be used at creation *)

Hash Table

- Access operators.

```
val add : ('a, 'b) t -> 'a -> 'b -> unit  
(* (add tbl x y) to add (x,y) to tbl *)
```

```
find : ('a, 'b) t -> 'a -> 'b  
(* (find tbl x) returns current binding of x *)
```

```
remove : ('a, 'b) t -> 'a -> unit  
(* (remove tbl x) remove current binding of x *)
```

Memoization

Memoization

- Redundant calls can be eliminated by tabulation using dynamic programming.
- General Technique : Memoization.
- Can use hash table to store previously computed value for retrieval rather than re-computation.

Recursion

- Recall naïve fibonacci.

```
let rec fib n =  
  
  if n<=1 then 1  
  
  else fib (n-1) + fib(n-2);;
```

- Contains many redundant fib calls when computed with moderately sized inputs.

Using Memo-Functions

- This stores previously computed values in say a hash table. Example:

```
let fib_hash = Hashtbl.create 10;;  
  
let rec fib_memo n =  
  if n<=1 then 1  
  else try  
    Hashtbl.find fib_hash n  
  with _ ->  
    let r = fib_memo (n-1) + fib_memo (n-2) in  
    let _ = Hashtbl.add fib_hash n r in  
    r
```

- Trade off memory space for time.

Pure Memo-Function

- Localize the effect of memo-table:

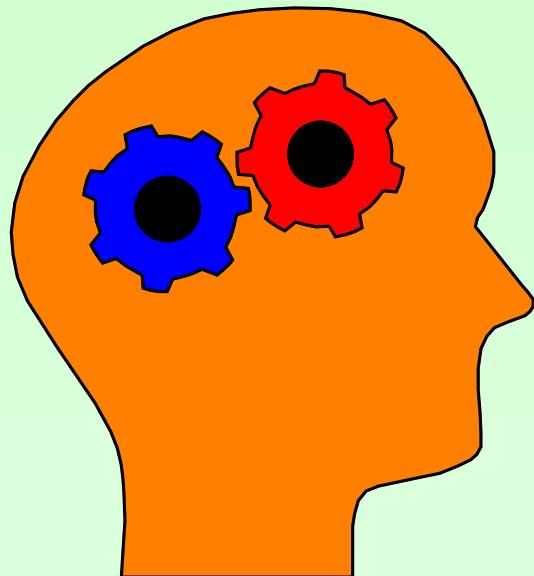
```
let fib_memo2 n =
  let tbl = Hashtbl.create 10 in
  let rec aux n =
    if n<=1 then 1
    else try
      Hashtbl.find tbl n
    with _ ->
      let r = aux (n-1) + aux (n-2) in
      let _ = Hashtbl.add tbl n r in
      r
  in aux n
```

- Pure function when viewed from outside.
Tradeoffs?



CS2104: Programming Languages Concepts

Lecture 11 : *OOP and Modules in OCaml*



“OOP and Modules”

Lecturer : [Chin Wei Ngan](#)
Email : chinwn@comp.nus.edu.sg
Office : COM2 4-32

Topics

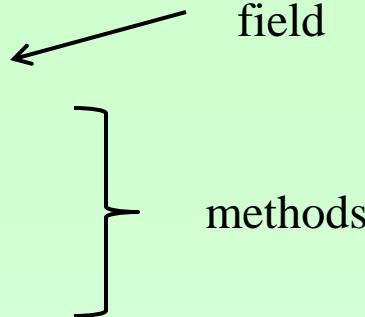
- Classes & Objects
- Structural Types and Row Polymorphism
- Modules : Structure and Signature
- ADT
- Functors

Classes and Objects

OCaml Class

- Each class comprise of fields and methods.

```
class counter =  
object  
  val mutable x = 0  
  method inc = x <- x + 1  
  method get = x  
  method set y = x <- y  
end;;
```



- Class is a factory of objects:

```
let p = new counter;;  
let q = new counter;;  
p # inc;;  
q # set 5;;
```

Object

- We can create a single object (without class), as follows.

```
let p =
  object
    val mutable x = 0
    method inc = x <- x + 1
    method get = x
    method set y = x <- y
  end;;
```

- This is similar to a singleton class in Scala, but is actually closer to an anonymous class.

Class Parameters

- Class may have parameters for its class constructors.

```
class counter init =
object
  val mutable x = init
  method inc = x <- x + 1
  method get = x
  method set y = x <- y
end;;
```

- Type of class constructor would be a function.

```
class counter : int -> object ... end
```

- Note that class name is abbreviation of object type itself.

Reference to Self/This

- You can explicitly name the current object.

```
class count_step init step =  
object (s) ← name for  
    val mutable x = 0  
    method inc = x <- x + step  
    method get = x  
    method print = string_of_int (s # get)  
end;;
```

- This is a named version of the current “this” object in Java/Scala.

Class Inheritance

- We can use class inheritance to obtain *fields* and *methods* of prior (super) class, and to support overriding.

```
class count_step init step =
  object (s)
    inherit counter init
    method inc = x <- x + step
    method print = string_of_int (s # get)
  end;;
```

Class Polymorphism

- We can support polymorphic classes using type variables.
- A simple generic buffer.

```
class ['a] buffer init =
object
  val mutable value : 'a = init
  method get = value
  method set n = value <- n
end;;
```

Class Signature

- Class type is based on structure of the set of *visible* methods.
- Type signature of generic buffer (*without* fields as they are always hidden).

```
class type ['a]  buffer_type =
object
  method get : 'a
  method set : 'a -> unit
end;;
```

Structural Type Equivalence

- Another equivalent generic buffer class is:

```
class ['a] buffer2 init =
object
  val mutable value : 'a = init
  val mutable is_empty = false
  method get =
    if is_empty then failwith "empty buffer"
    else (is_empty <- true; value)
  method set n = (value <- n; is_empty <- false)
  method private reuse = is_empty <- false
end;;
```

- This has the same set of visible methods (with the same type) as the earlier `buffer_type`.

Structural Typing

- Two types are the same if they are *structurally equivalent* to each other on the *visible* methods.

```
let foo (v:'a buffer_type) = v # get;;
(* foo : 'a1 buffer_type -> 'a1 *)
```

```
let v = new buffer 5;;
```

```
let w = new buffer2 5;;
```

- Though `v` and `w` are created by different classes, they have the *same* type signature.

Subtyping via Row Polymorphism

- Structural subtyping is supported via “*row polymorphism*”.

```
let foo2 (v) = v # get;;
(* foo2 : <get : 'b; ... >-> 'b *)
```

```
let v = new buffer 5;;
let w = new buffer2 5;;
foo2 v;;
foo2 z;;
```

- The row polymorphic type `<get : 'b; ... >` will unify with any type with a `get` method in its class type.
- The matched type need not be structurally equivalent.

Subtyping via Coercion

- It is possible to coerce type via up-cast operation.

```
let foo3 (v) = (v:>'b buffer) # get;;
(* foo3 : 'b #buffer -> 'b *)
```

```
let v = new buffer 5;;
let w = new buffer2 5;;
foo3 v;;
foo3 z;;
```

- The will unify with any type that contains at least the visible methods from the buffer class.
- The matched type is a structural subtype.

Object Cloning

- An object can be cloned using `Oo.copy` which will make a new instance of its field variables. This is a *shallow* copy.

```
let foo2 (v) = v # get;;
let w = new buffer2 5;;
let z = Oo.copy w;;
(* Oo.copy : (< .. > as 'a) -> 'a *)
foo2 v;;
foo2 z;;
```

- Deeper sharing are still present in the copied object.
- Note that `Oo.copy` works with any object type `< .. >`.

Functional Objects

- If we *disallow* mutability of fields, we can get *functional* objects.

```
class func_point y =
object
  val x = y
  method move (d:int) : func_point =
    new func_point (x+d)
end;;
```

- Note that this is also a *recursive* class type. What is its class type signature?

Mutually-Recursive Classes

- Mutual-recursive types can also be defined.

```
class window =
  object
    val mutable top_widget
      = (None : widget option)
    method top_widget = top_widget
  end
and widget (w : window) =
  object val window = w
    method window = window
  end;;
```

- The class signature would be mutual-recursive too.

Virtual Methods and Classes

- We can define a class with some *undefined* methods.

```
class virtual ['a]  buffer_eq init =
  object (this)
    val mutable value : 'a = init
    method get = value
    method virtual eq : 'a buffer_eq -> bool
    method neq b = not(this # eq b)
  end;;
```

- Objects of virtual (undefined) classes cannot be instantiated.
- This is the same as abstract classes and abstract methods in Scala.

Implementing Virtual Methods

- Concrete classes should give definitions for its virtual methods.

```
class ['a] buffer init =
  object (this)
    inherit ['a] buffer_eq init
    method eq that
      = this # get = that # get
    method set n = value <- n
  end;;
```

- Class instantiation now possible with concrete classes.

```
let b = new buffer 5
let c = new buffer_eq 5 (* invalid *)
```

Modules & Functors

Modules

- OCaml has an *advanced* module system.
- Modules (like Scala/Java package) used to structure large programs.
- Modules allow us to conserve name space.
- Modules allow us to support ADT.

Module

- Modules can be used to group *types, values, functions, exceptions* and other *modules* together.

```
module Buffer =
  struct
    type 'a t = ('a option) ref
    let emp () : 'a t = ref None
    let get buf = match !buf with
      | None -> failwith "Buffer_Err"
      | Some r -> (buf := None; r)
    let put buf x = buf := Some x
  end
```

- A *structure* is an implementation for module.

Qualified Names

- We use qualified names to access values.

```
type 'a tt = ' a Buffer.t
let g = Buffer.get
```

- However, we may open module to have its entities visible locally, without any qualifiers.

```
open Buffer
type 'a tt2 = ' a t
let g = get
```

Module Signature

- Each module has a type signature that can also be explicitly declared. For example:

```
module type BUFFER =
sig
  type 'a t = ('a option) ref
  exception Buffer_Err
  val emp : unit -> 'a t
  val get : 'a t -> 'a
  val put : 'a t -> 'a -> unit
end
```

- Convention to write module type entirely in upper-case.
- A type signature is an interface for a module.

Module ADT

- We can hide implementation details by using a more *abstract* type signature.

```
module type BUFFER_ABS =
sig
  type 'a t
  val emp : unit -> 'a t
  val get : 'a t -> 'a
  val put : 'a t -> 'a -> unit
end
```

- Type `'a t` is now made abstract with two implementation details hidden
 - (i) option ref
 - (ii) Buffer_Err exception

Module ADT

- Type of ADT is actually an existential type.

```
module type BUFFER_ABS =
sig
  type 'a t
  val emp : unit -> 'a t
  val get : 'a t -> 'a
  val put : 'a t -> 'a -> unit
end
```

Similar to existential type:

$$\exists t. \{ \text{emp} : \text{unit} \rightarrow 'a t; \\ \text{get} : 'a t \rightarrow 'a; \\ \text{put} : 'a t \rightarrow 'a \rightarrow \text{unit} \}$$

Abstract Data Type Implementation

- We can provide implementation for abstract modules.

```
module BufferADT = Buffer : BUFFER_ABS
```

- Without information hiding, we can access more things.

```
let b1 = Buffer.Buffer_Err  
let b2 = !(Buffer.emp ()) == None
```

- Using abstract module **BufferADT**, we disallow implementation details to be exposed

```
let b1 = BufferADT.Buffer_Err (* invalid *)  
let b2 = !(BufferADT.emp ()) == None (* invalid *)
```

Abstract Data Type Implementation

- Module implementation can be directly *associated* with ADT type.

```
module Buffer2 : BUFFER_ABS  =
  struct
    type 'a t = ('a option) ref
    exception Buffer_Err
    let emp () : 'a t = ref None
    let get buf = match !buf with
      | None -> raise Buffer_Err
      | Some r -> (buf := None; r)
    let put buf x = buf := Some x
  end
```

Alternative ADT Implementation

- We can choose other kinds of implementation, such as unbounded *mutable list* for our buffers.

```
module BufferL : BUFFER_ABS =
  struct
    type 'a t = ('a list) ref
    let emp () = ref []
    let get buf = match !buf with
      | [] -> failwith "empty"
      | r::xs -> (buf := xs; r)
    let put buf x = buf := (!buf@[x])
  end
```

Functor

- Functors are functions from structures to structures.
- Functor for priority buffer.

```
module Buffer_P =
  functor (Elt: PRIORITY_TYPE) ->
    struct
      type element = Elt.t
      type t = (element list) ref
      :
      let put buf x = buf := ins !buf x (Elt.get_p x)
    end;;
```

Module Type for Priority

- Module Type.

```
module type PRIORITY_TYPE =
sig
  type t
  val get_p : t -> int
end;;
```
- Example 1 :

```
module Int : PRIORITY_TYPE =
  struct
    type t = int
    let get_p x = x
  end;;
```
- Example 2 :

```
module Int_P : PRIORITY_TYPE =
  struct
    type elm
    type t = elm * int
    let get_p (_,x) = x
  end;;
```

Usage of Functors

- Specializing two different modules with Functors

```
module PQ1 = Buffer_P(Int)
```

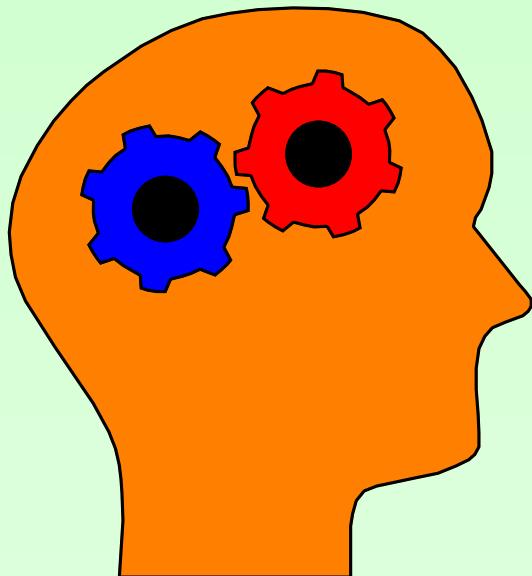
```
module PQ2 = Buffer_P(Int_P)
```

- Notice that Module generated from Functor application.
- Supports code reuse via modules.



CS2104: Programming Languages Concepts

Lecture 12-13 : Scala Highlights



*“Boosting Java
With FP and Stronger Types”*

Lecturer : Chin Wei Ngan
Email : chinwn@comp.nus.edu.sg
Office : COM2 4-32

Motivation for Scala Language

- interoperable with Java
- conciseness (2-10 times shorter)
- supports higher-order functions (OOP+FP)
- advance static typing and inference
- developed by Martin Odersky group @ EPFL

Conciseness

```
// in Java
class MyClass {
    private int index;
    private String name;
    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

```
// in Scala:
class MyClass(index: Int, name: String)
```

Object-Oriented Style Supported

- Every value is an object
- Types and behavior of objects are described by
 - classes (including abstract classes)
 - *traits*
- Class abstraction are extended by
 - sub-classing
 - *mixin* composition
(cleaner replacement for multiple inheritance)

Functional Style Supported

- Every function is an object
- Functions are first-class values
 - passed as argument
 - returned as result
 - can be stored in data structures
- Anonymous functions allowed
- Pattern-matching via Case Classes

Highlights of Scala

- Scala Classes
- Types
- Higher-Order Functions
- Lists
- Pattern-Matching
- Traits as Mixins
- Implicits

Scala Classes

Scala Classes

- Factory templates that can be instantiated to objects.
- Class Parameters and Explicit Overriding

```
class Point(xc: Int, yc: Int) {  
    var x: Int = xc  
    var y: Int = yc  
    def move(dx: Int, dy: Int) {  
        x = x + dx  
        y = y + dy  
    }  
    override def toString(): String  
        = "(" + x + ", " + y + ")";  
}
```

Scala Classes

- Parameterised by constructor arguments.
Objects are instantiated with new command, e.g.

```
new Point(3,4);
```

- Uses dynamically-dispatched methods only:

```
this.move(dx,dy);  
this.toString();
```

Scala Classes

- A class with a single object is declared with the "object" keyword.
- This example captures an executable application with a main method that can be directly executed.

```
object Classes {  
    def main(args: Array[String]) {  
        val pt = new Point(1, 2)  
        println(pt)  
        pt.move(10,10)  
        println(pt)  
    }  
}
```

Abstract Classes

- Classes are parameterized with values and with types.
- Supports generic classes.
- Abstract class may have
 - (i) deferred/abstract type
 - (ii) deferred value definition

```
abstract class Buffer {  
    type T  
    val element: T  
}
```

Abstract Classes

- Can *reveal* more information on an abstract type by giving *type bounds*.

```
abstract class SeqBuffer extends Buffer {  
    type U  
    type T <: Seq[U]  
    def length = element.length  
}
```

- *Refinement* could be added to instantiate abstract type definition:

```
abstract class IntSeqBuffer extends SeqBuffer {  
    type U = Int  
}
```

Abstract Classes

- Abstract type definition can be turned into type parameters with declaration-site variance annotation:

```
abstract class Buffer[+T] {  
    val element: T  
}
```

```
abstract class SeqBuffer[U,+T<:Seq[U]] extends Buffer[T] {  
    def length = element.length  
}
```

Types

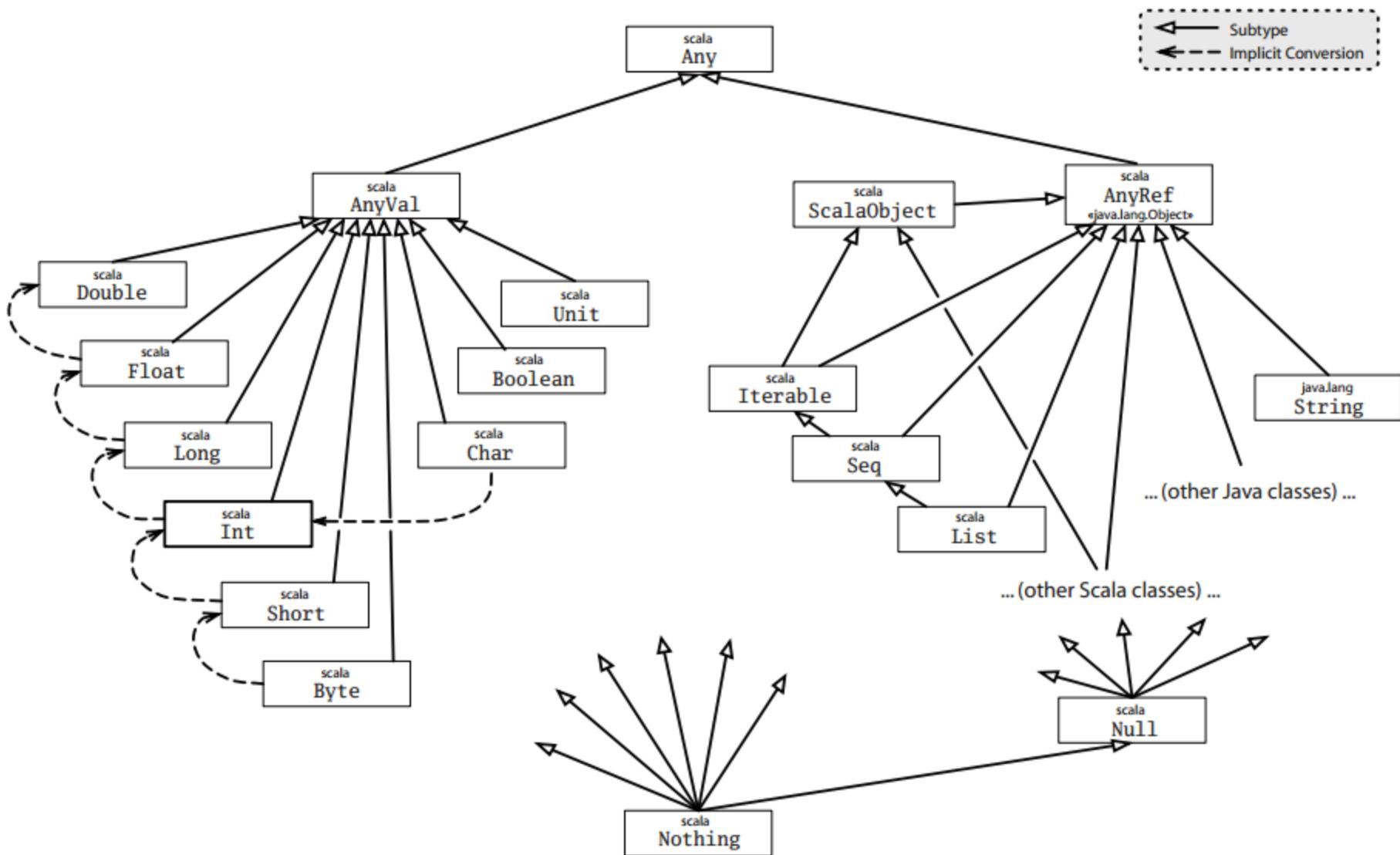


Figure 11.1 · Class hierarchy of Scala.

Unified Types

- all values (including numerics and functions) are objects
 - all values are instances of a class
 - superclass of all classes

`scala.Any`

`:> scala.AnyVal`

`:> scala.AnyRef`

- Every user-defined class
 - are (indirectly) subclass of `scala.AnyRef`
 - implicitly extends trait `scala.ScalaObject`

Unified Types

```
object UnifiedTypes {
    def main(args: Array[String]) {
        val set = new scala.collection.mutable.HashSet[Any]
        set += "This is a string"    // add a string
        set += 732                  // add a number
        set += 'c'                  // add a character
        set += true                 // add a boolean value
        set += main _              // add the main function
        val iter: Iterator[Any] = set.iterator
        while (iter.hasNext) {
            println(iter.next.toString())
        }
    }
}
```

```
c
true
<function>
732
This is a string
```

Polymorphic Methods

- Methods can be parameterized by both values and types
- Values are enclosed in parenthesis, while types are declared within a pair of brackets.

```
object PolyTest extends Application {  
    def dup[T](x: T, n: Int): List[T] =  
        if (n == 0) Nil  
        else x :: dup(x, n - 1)  
    println(dup[Int](3, 4))  
    println(dup("three", 3))  
}
```

Lists

Pervasive List

- List is a pervasive data type in programming. Though Array also captures sequence, List has the following properties
 - (i) immutable
 - (ii) unbounded length
 - (iii) dynamically-linked data structure
- Examples:

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```

Pervasive List

- Two basic constructors (i) **Nil** (ii) **::**

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) :: 
             (0 :: (1 :: (0 :: Nil))) :: 
             (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

- Three primitive List operations:
 - **head** returns the first element of a list
 - **tail** returns a list consisting of all elements except the first
 - **isEmpty** returns true if the list is empty

Pervasive List

Extractor is used to provide List patterns/views:

```
scala> val List(a, b, c) = fruit
a: String = apples
b: String = oranges
c: String = pears
```

```
scala> val a :: b :: rest = fruit
a: String = apples
b: String = oranges
rest: List[String] = List(pears)
```

Pervasive List

How would you *join* two Lists together?

```
def append[T] (xs: List[T], ys: List[T]): List[T] =  
  xs match {  
    case List()    =>  
    case x :: xs1 =>  
  }
```

How would you *reverse* a given List?

```
def rev[T] (xs: List[T]): List[T] =  
  xs match {  
    case List()    =>  
    case x :: xs1 =>  
  }
```

Folding over List

- Common to combine the elements of a list with some operator. For instance:

```
sum(List(a, b, c)) equals 0 + a + b + c
```

This summation can be implemented by:

```
def sum(xs: List[Int]): Int  
= (0 /: xs) (_ + _)
```

- Similarly:

```
product(List(a, b, c)) equals 1 * a * b * c
```

This product operation can be implemented by:

```
def product(xs: List[Int]): Int = (1 /: xs) (_ * _)
```

Folding over Lists

- Fold left operation :

```
(z : List(a, b, c)) (op)
  equals op(op(op(z, a), b), c)
```

- Fold right operation:

```
(List(a, b, c) :\ z) (op)
  equals op(a, op(b, op(c, z)))
```

Pattern Matching

Pattern Matching

- Match on any sort of data with a *first-match* policy
- Match keyword allows *pattern-matching function* to be applied to an object, e.g.

```
object MatchTest1 extends Application {  
    def matchTest(x: Int): String = x match {  
        case 1 => "one"  
        case 2 => "two"  
        case _ => "many"  
    }  
    println(matchTest(3))  
}
```

Pattern Matching

- Possible to match a value against patterns of different types. e.g.

```
object MatchTest2 extends Application {  
    def matchTest(x: Any): Any = x match {  
        case 1 => "one"  
        case "two" => 2  
        case y: Int => "scala.Int"  
    }  
    println(matchTest("two"))  
}
```

Case Classes

- Case classes allow their constructor parameters to be exported via pattern-matching.
- Example to denote untyped lambda calculus:

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

Case Classes

- Advantages : "new" primitive is not required

```
Fun("x", Fun("y", App(Var("x"), Var("y")))))
```

- Constructor parameters are treated as public values that can be directly accessed

```
val x = Var("x")
Console.println(x.name)
```

- Automatic derivation of *equality* and *toString* method.

```
val x1 = Var("x")
val x2 = Var("x")
val y1 = Var("y")
println("") + x1 + " == " + x2 + " => " + (x1 == x2)
println("") + x1 + " == " + y1 + " => " + (x1 == y1))
```

Case Classes

- Supports pattern-matching:

```
// pretty printer
object TermTest extends Application {
    def printTerm(term: Term) {
        term match {
            case Var(n) =>
                print(n)
            case Fun(x, b) =>
                print("^ " + x + ".")
                printTerm(b)
            case App(f, v) =>
                Console.print("(")
                printTerm(f)
                print(" ")
                printTerm(v)
                print(")")
        }
    }
}
```

Higher-Order Functions

Higher-Order Functions

- Scala supports functions as *first-class* values
 - function as parameter
 - function as result
 - function inside data structure.
- An example:

```
def apply(f: Int => String, v: Int) = f(v)
```

Higher-Order Functions

```
class Decorator(left: String, right: String) {  
    def layout[A](x: A) = left + x.toString() + right  
}  
  
object FunTest extends Application {  
    def apply(f: Int => String, v: Int) = f(v)  
    val decorator = new Decorator("[", "]")  
    println(apply(decorator.layout, 7))  
}
```

- Polymorphic layout of type `A => String` is automatically coerced to a value of type `Int => String`

Anonymous Functions

- A shorthand for writing anonymous functions

```
(x: Int) => x + 1
```

- Full longer form :

```
new Function1[Int, Int] {  
    def apply(x: Int): Int = x + 1  
}
```

- Function with two parameters

```
(x: Int, y: Int) => "(" + x + ", " + y + ")"
```

- Function with no parameter

```
() => { System.getProperty("user.dir") }
```

Placeholder Function

- The symbol `_` denotes a placeholder for parameters

`_ + 1` or `(_:Int)+1`

denotes :

`(x: Int) => x + 1`

- Similarly:

`(_:Int) + (_:int)`

denotes :

`(x:Int,y:Int) => x + y`

Types for Functions

- Shorthand for Types:

```
Int => Int  
(Int, Int) => String  
() => String
```

- Longer form for Types

```
Function1[Int, Int]  
Function2[Int, Int, String]  
Function0[String]
```

Operators

- Infix and postfix form are occasionally more readable
 - Method with one parameter → infix
 - Method with no parameter → postfix.

```
class MyBool(x: Boolean) {  
    def and(that: MyBool): MyBool = if (x) that else this  
    def or(that: MyBool): MyBool = if (x) this else that  
    def negate: MyBool = new MyBool(!x)  
}
```

Operators

- Possible to use "negate" in postfix form:

```
def not(x: MyBool) = x negate;  
    // semicolon required here
```

- Possible to use "and" and "or" in infix form:

```
def xor(x: MyBool, y: MyBool)  
    = (x or y) and not(x and y)
```

- Traditional form:

```
def not(x: MyBool) = x.negate;  
    // semicolon required here  
def xor(x: MyBool, y: MyBool)  
    = x.or(y).and(x.and(y).negate)
```

Currying

- Methods may define multiple parameter lists.
- When a method is called with fewer argument list, it yields a function which expects the remaining parameter lists.

```
object CurryTest extends Application {  
    def filter(xs: List[Int], p: Int => Boolean): List[Int] =  
        if (xs.isEmpty) xs  
        else if (p(xs.head)) xs.head :: filter(xs.tail, p)  
        else filter(xs.tail, p)  
    def modN(n: Int)(x: Int) = ((x % n) == 0)  
    val nums = List(1, 2, 3, 4, 5, 6, 7, 8)  
    println(filter(nums, modN(2)))  
    println(filter(nums, modN(3)))  
}
```

```
filter : (List[Int], Int => Boolean) => List[Int]  
modN : Int => (Int => Boolean)
```

Traits as Mixin

Traits

- Similar to interfaces in Java
 - can have fields and methods
 - may have default implementation from some methods
 - do not have constructor parameters

Example:

```
trait Similarity {  
    def isSimilar(x: Any): Boolean  
    def isNotSimilar(x: Any): Boolean = !isSimilar(x)  
}
```

`isSimilar` is an abstract method,
but `isNotSimilar` has a concrete implementation

Traits

```
class Point(xc: Int, yc: Int) extends Similarity {
    var x: Int = xc
    var y: Int = yc
    def isSimilar(obj: Any) =
        obj.isInstanceOf[Point] &&
        obj.asInstanceOf[Point].x == x
}
object TraitsTest extends Application {
    val p1 = new Point(2, 3)
    val p2 = new Point(2, 4)
    val p3 = new Point(3, 3)
    println(p1.isNotSimilar(p2))
    println(p1.isNotSimilar(p3))
    println(p1.isNotSimilar(2))
}
```

Mixin Class Composition

- Neither single inheritance, nor just multiple inheritance.
- Allows reuse of new member definitions from a class
- An abstract class:

```
abstract class AbsIterator {  
    type T  
    def hasNext: Boolean  
    def next: T  
}
```

Mixin Class Composition

- A mixin through keyword trait:

```
trait RichIterator extends AbsIterator {  
    def foreach(f: T => Unit)  
        { while (hasNext) f(next) }  
}
```

- A concrete iterator class (where type T has been instantiated):

```
class StringIterator(s: String) extends AbsIterator {  
    type T = Char  
    private var i = 0  
    def hasNext = i < s.length()  
    def next = { val ch = s.charAt(i); i += 1; ch }  
}
```

Mixin Class Composition

- A mixin class composition
(with both `StringIterator` and `RichIterator`)

```
object StringIteratorTest {  
    def main(args: Array[String]) {  
        class Iter extends StringIterator(args(0))  
            with RichIterator  
        val iter = new Iter  
        iter foreach println  
    }  
}
```

Implicits

Implicit Parameters

- Implicit can help with type conversion.
- For example, `Double` cannot be automatically converted to `Int`.

```
scala> val i: Int = 3.5
<console>:4: error: type mismatch;
 found : Double(3.5)
 required: Int
val i: Int = 3.5
```

Implicit Parameters

However, we can define an *implicit* conversion to automatically perform such casting.

```
scala> implicit def doubleToInt(x: Double)
           = x.toInt
doubleToInt: (x: Double) Int
scala> val i: Int = 3.5
i: Int = 3
```

Implicit Parameters

- Implicit allows our code to inter-operate with new types:

```
class Rational(n: Int, d: Int) {  
    ...  
    def + (that: Rational): Rational = ...  
    def + (that: Int): Rational = ...  
}
```

- Add an implicit to convert to Rational type:

```
implicit def intToRational(x:Int) =  
    new Rational(x,1)
```

Implicit Parameters

- Default parameter can be customized using `implicit`.

```
class PreferredPrompt(val preference: String)
object Greeter {
  def greet(name: String)(implicit prompt: PreferredPrompt) {
    println("Welcome, " + name + ". The system is ready.")
    println(prompt.preference)
  }
}
```

Implicit parameter can be supplied explicitly:

```
scala> val bobsPrompt = new PreferredPrompt("relax> ")
scala> Greeter.greet("Bob")(bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```

Implicit Parameters

- We can also define it implicitly:

```
object JoesPrefs {  
    implicit val prompt =  
        new PreferredPrompt("Yes, master> ")  
}
```

- This results in :

```
scala> import JoesPrefs._  
import JoesPrefs._  
scala> Greeter.greet("Joe")  
Welcome, Joe. The system is ready.  
Yes, master>
```

Implicit Parameters

- a special feature to support systematic method overloading
- define abstract classes

```
abstract class SemiGroup[A] {  
    def add(x: A, y: A): A  
}  
abstract class Monoid[A] extends SemiGroup[A] {  
    def unit: A  
}
```

Implicit Parameters

- allow instances of these abstract classes.
e.g. `Int` and `String` are indirectly instances of `Monoid[?]`

```
object ImplicitTest extends Application {  
    implicit object StringMonoid extends Monoid[String] {  
        def add(x: String, y: String): String = x concat y  
        def unit: String = ""  
    }  
    implicit object IntMonoid extends Monoid[Int] {  
        def add(x: Int, y: Int): Int = x + y  
        def unit: Int = 0  
    }  
}
```

Implicit Parameters

- with implicit objects, we can now define generic method, that sum up a list of monoid values:
e.g.

```
def sum[A] (xs: List[A]) (implicit m: Monoid[A]): A =  
  if (xs.isEmpty) m.unit  
  else m.add(xs.head, sum(xs.tail))
```

- implicit parameters can be inferred given:

```
println(sum(List(1, 2, 3)))  
println(sum(List("a", "b", "c")))
```

can infer:

```
println(sum(List(1, 2, 3)) (IntMonoid))  
println(sum(List("a", "b", "c")) (StringMonoid))
```

Miscellaneous

Packages

- A package is a special object which defines a set of member classes, objects and packages.
- A packaging package `p { ds }` injects all definitions in `ds` as members into the package whose qualified name is `p`.
- If a definition in `ds` is labeled *private*, it is visible only for other members in the package.
- A *protected* modifier allows its members to be accessible from all code inside the package `p`.

Import

- An import clause determines a set of names of that can be used without qualifications.

```
import p._  
  //all members of p  
  //(this is analogous to import p.* in Java).  
import p.x  
  //the member x of p.  
import p.{x => a}  
  //the member x of p renamed as a.  
import p.{x, y}  
  //the members x and y of p.  
import p1.p2.z  
  //the member z of p2, itself member of p1.
```

Import

- Implicitly imported into every compilation unit:
 - the package `java.lang`,
 - the package `scala`,
 - and the object `scala.Predef`.

Type Inference

Local Type Inference

- Types can either be declared or inferred.
- Eases programmers burden by automatically inferring certain type annotations.
- Can infer type of:
 - (i) variable (through its initialization)
 - (ii) results of non-recursive method
 - (iii) type instantiation of polymorphic methods
- May fail occasionally.

Local Type Inference

Example 1: Type Instantiation of Generic Methods

```
case class MyPair[A, B](x: A, y: B);
object InferenceTest3 extends Application {
  def id[T](x: T) = x
  val p = new MyPair(1, "scala")
    // type: MyPair[Int, String]
  val q = id(1)
    // type: Int
}
```

Explicit Instantiation:

```
val x: MyPair[Int, String] =
  new MyPair[Int, String](1, "scala")
val y: Int = id[Int](1)
```

Local Type Inference

Example 2:

```
object InferenceTest2 extends Application {  
    val x = 1 + 2 * 3          // the type of x is Int  
    val y = x.toString()        // the type of y is String  
    def succ(x: Int) = x + 1   // method succ returns  
                            // Int values  
}
```

Failures of type inference

```
object InferenceTest3 {  
    def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)  
}  
  
object InferenceTest4 {  
    var obj = null    // Null inferred  
    obj = new Object()  
}
```

Runtime Type Representation

- `classOf[T]` returns string representation of a type
- `var.getClass()` returns the representation of runtime type for object

```
object ClassReprTest {  
    abstract class Bar {  
        type T <: AnyRef  
        def bar(x: T) {  
            println("5: " + x.getClass() ) }  
    }  
    def main(args: Array[String]) {  
        println("1: " + args.getClass())  
        println("2: " + classOf[Array[String]])  
        new Bar {  
            type T = Array[String]  
            val x: T = args  
            println("3: " + x.getClass() )  
            println("4: " + classOf[T])  
            }.bar(args) }  
}
```