Lecture 3 : **Data Types and Control Constructs**
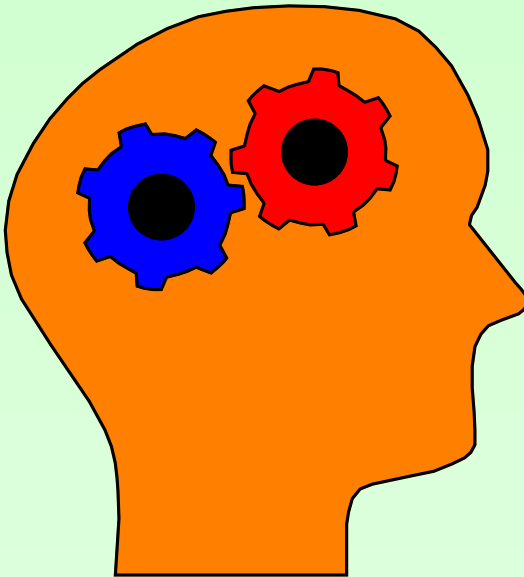
"Types, Constructs and Recursion and Type Classes"

Lecturer : <u>Chin</u> 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
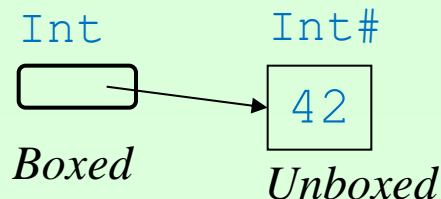
Primitive Types

- 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`

```
Int          Int#
┌──────┐     ┌──────┐
│  ◯   │────▶│  42  │
└──────┘     └──────┘
 Boxed       Unboxed
```

- 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.

```scala
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]
};
```

Int Float

Char[20]
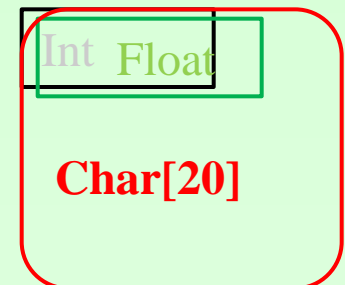
Algebraic Data Type (in Haskell)

- 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
```

- 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;
```
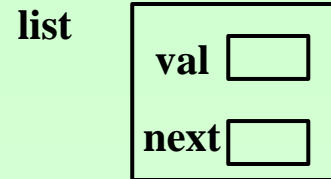
count	4
init	6

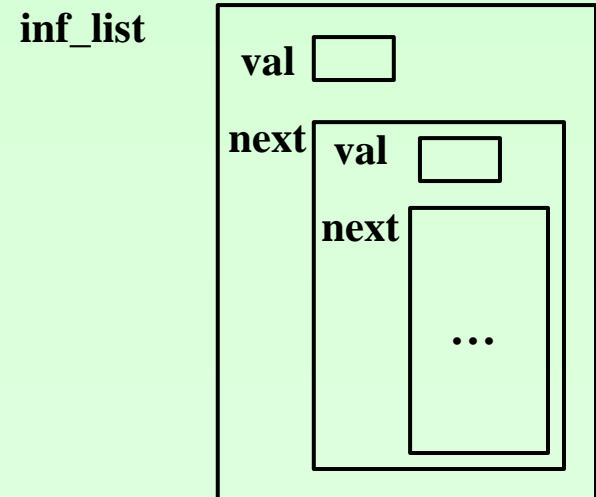| ptr	

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

- Pointer types are crucial for recursive data type.

- An example in C is:

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

list

val	☐
next	☐

- We cannot use below. Why?

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

inf_list

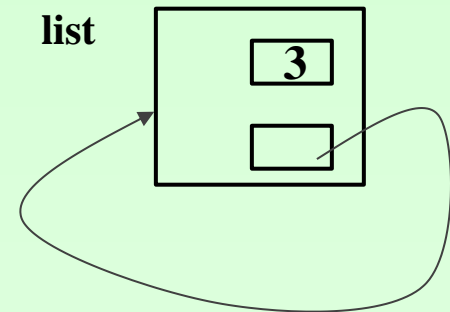|	
val	☐
next	**val** ☐
	next ...

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)
```

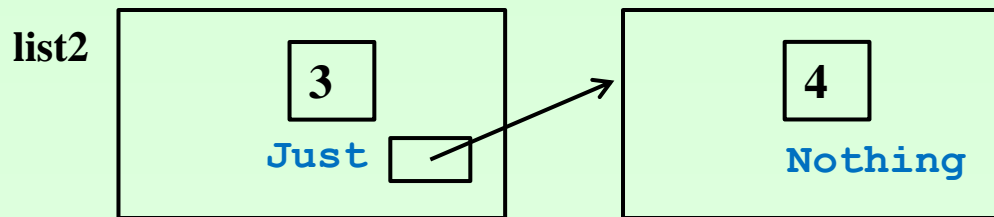- We may use a general Maybe type

```
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))
```

list2

3		4
Just		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 <= y && x /= y
x >= y        = y <= x
x > y         = y < x
min x y       = if x<=y then x else y
max x y       = if x>=y then x 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 :**

```haskell
    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.**

```haskell
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 "Divison by zero"
myDiv3 x y = Right (x / y)
```
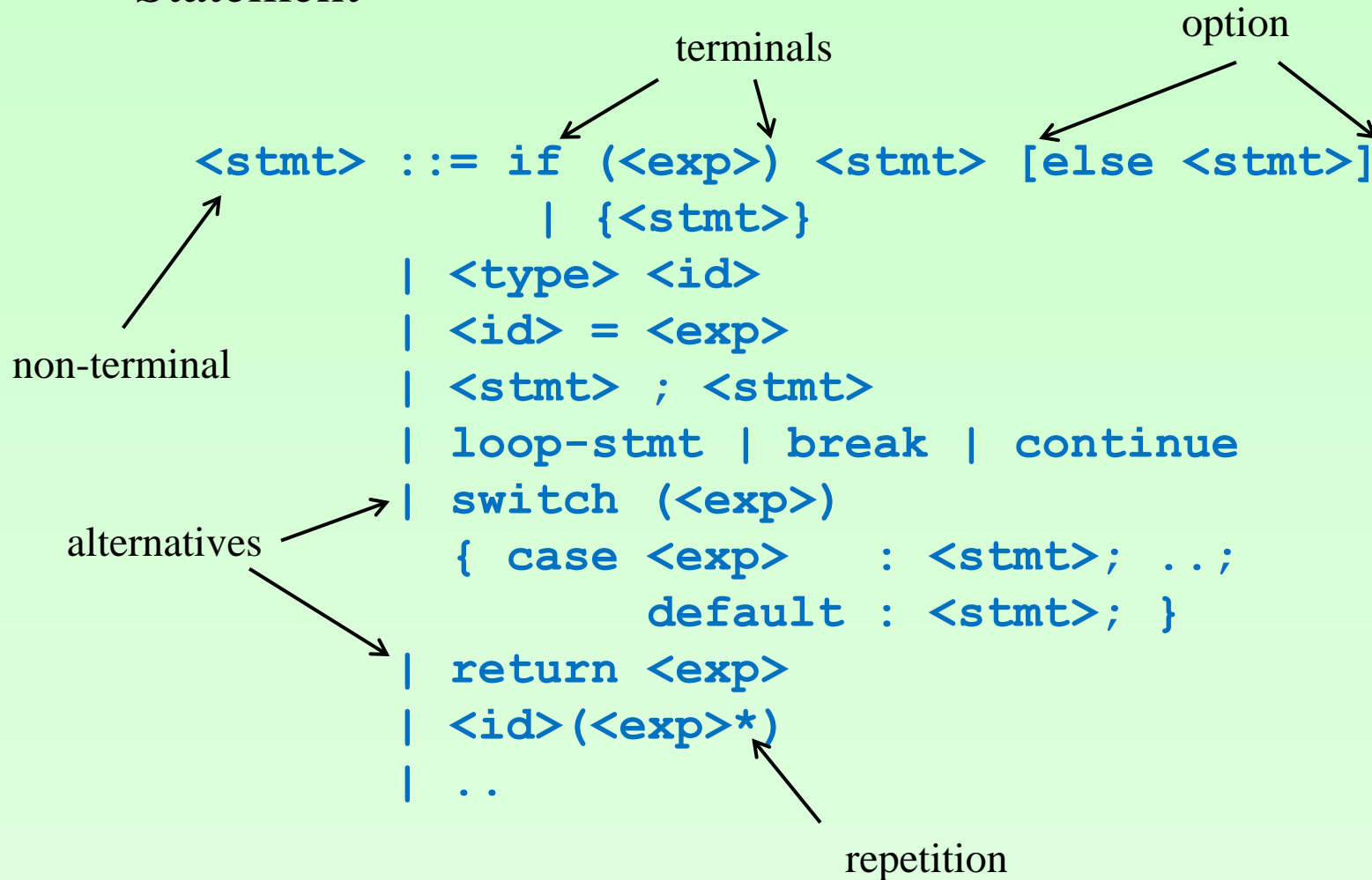
Language Constructs and Recursion

- 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

- Statement

terminals   option

```
<stmt> ::= if (<exp>) <stmt> [else <stmt>]
           | {<stmt>}
        | <type> <id>
        | <id> = <exp>
        | <stmt> ; <stmt>
        | loop-stmt | break | continue
        | switch (<exp>)
        { case <exp>    : <stmt>; ..;
                default : <stmt>; }
        | return <exp>
        | <id>(<exp>*)
        | ..
```

non-terminal

alternatives

repetition

- 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>
```
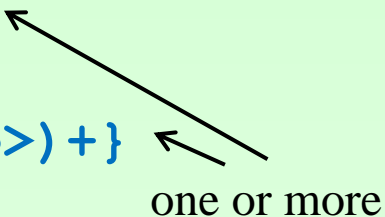
- Expression-Oriented Constructs

```
<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