Branching operations

All programming languages have branching operations.

You know ‘em. You love ‘em.

• if
• switch
• while
If (e) S1 else S2

evaluates e to a value v
if v is true, do S1
if v is false, do S2
Switch

switch (i) {
    case 0: assert(i == 0); break;
    case 1: assert(i == 1); break;
    default: assert(i != 0 && i != 1); break;
}
Switch implementation

Implemented with a jump table.

switch (x) {
    case 5: S1
    case 6: S2
    case 8: S3
    default: S4
}

table[0] = L1
table[1] = L2
table[2] = L4  // default
table[3] = L3

t = x - 5
if (t < 0) goto L4
if (t >= 4) goto L4
goto table[t]

L1: S1
L2: S2
L3: S3
L4: S4
Can also branch on a type. In Java this is done with instanceof, which returns a boolean.

```java
if (x instanceof C) {
    ...
}
```
**Instanceof + cast**

Typical pattern:

```java
if (e.f instanceof C) {
    // careful: other thread might do e.f = new B
    C x = (C) e.f;
    ...
}
```

Note: this tests the type of x twice!

In multithreaded programs, x might change between the two tests, causing the cast to fail.
**Typecase**

Branches on the type of a value.
Only one test is performed, avoiding data race problem.

```plaintext
typecase (e) {
    case A x: ...
    case B y: ...
}
```

similar to this Java code (but often faster since can use a jump table):

```plaintext
z = e;
if (z instanceof A) { A x = (A) z; ... }
else if (z instanceof B) { B y = (B) z; ... }
```
Pattern matching

Found in most functional languages

Based on algebraic datatypes
Algebraic data types

The primary way to build new data abstractions in the ML family of languages is the algebraic data type.

Binary trees in ML
- type ‘a tree = Leaf of ‘a | Node of ‘a tree * ‘a tree
- a tree of a’s is
  - either a leaf with an a
  - or, an interior node with two subtrees of a’s

Why “algebraic”?
In math, an “algebraic structure” is a set closed under one or more operations (e.g., addition, multiplication)
- Examples: groups, rings, fields, vector spaces

An algebraic data type is built from products (*) and sums (|)
Product types

AKA records, pairs, tuples

In ML, Haskell
- \( S \times T \times U \)

In Scala
- \((S, T, U)\)

Similar to a record or struct with fields named 1, 2, 3

```plaintext
struct A { S s; T t; U u; }
```
Arrays vs. tuples

Arrays: all elements are of the same type

```scala
scala> val A = Array(1,2,3)
A: Array[Int] = Array(1, 2, 3)

scala> val B = Array("hello", 1)
B: Array[Any] = Array(hello, 1)

scala> B(1)
res1: Any = 1
```

Tuples: elements can have heterogeneous types

```scala
scala> val p = ("hello", 1)
p: (java.lang.String, Int) = (hello,1)

scala> p._2
res3: Int = 1
```
Algebraic data types

A type with one or more variants
AKA disjoint union, tagged union

In Haskell:
• `data IntList = Nil | Cons Int IntList`

In ML:
• `datatype intlist = Nil | Cons of int * intlist`

In OCaml:
• `type intlist = Nil | Cons of int * intlist`

A value of type `IntList` is either a `Nil` or a `Cons`

Nil and Cons are called constructors (do not confuse)
Case classes

Scala can simulate ADTs with case classes:

- class IntList
- case class Nil extends IntList
- case class Cons(head: Int, tail: IntList) extends IntList
Case classes

case class Cons(head: Int, tail: IntList) extends IntList

declares a final subclass Cons of IntList
• with public final fields head and tail
• with a constructor that initializes the fields
• with structural equality for ==

• plus: don’t have to say ‘new’ to create an instance
Parametric polymorphism

ADTs support *parametric polymorphism*

- think generics like in Java or C#

list of `Int`
- `data IntList = Nil | Cons Int IntList`

list of ‘a’
- `data List a = Nil | Cons a (List a)`

All elements must be of the same type
- `[1,2,3]` is a `List int`
- `[“hi”, “mom”]` is a `List string`
- `[0, “mom”]` is an error
Parametric polymorphic case classes

list of Int
- class IntList
- case class Nil extends IntList
- case class Cons(head: Int, tail: IntList) extends IntList

list of ‘A’
- class List[A]
- case class Nil[A] extends List[A]
- case class Cons[A](head: A, tail: List[A]) extends List[A]
Actual definition of List in Scala

- sealed abstract class List[+A]
- case object Nil extends List[Nothing]
- case class ::(head: A, tail: List[A]) extends List[A]
Option types

One of the simplest ADTs e.g., in Haskell:

- \( \text{data Option } a = \text{Some } a \mid \text{None} \)

AKA Maybe

- \( \text{data Maybe } a = \text{Just } a \mid \text{Nothing} \)

Often used where \texttt{null} would be used in an OO language
Scala HashMap.get

`def get(key: Key): Option[Value]` allows you to distinguish between a null value and not present.
Binary trees

data Tree a = Empty | Node (Tree a) a (Tree a)
ADTs vs. objects

Algebraic data types define the *structure* of data, but do not define operations on the data

**Do not** hide representation of data
Extensibility

Objects

• easy to add new variants
  • subclassing

• hard to add new operations
  • add method to base class, implement in all subclasses, recompile everything

ADTs

• easy to add new operations
  • add a new function

• hard to add new variants
  • redefine data type, modify all matches that use that data type
Pattern matching

Given an ADT, can \textit{deconstruct} the value using pattern matching

\begin{verbatim}
fun sum xs = 
case xs of 
    Nil => 0  
    | Cons(y, ys) => y + (sum ys) 
end
\end{verbatim}

Essence of a pattern match is \textit{branch} and \textit{bind}

\begin{itemize}
    \item \textit{branches} on the structure of the value
    \item \textit{binds} variables to components of the matched structure
\end{itemize}
Pattern matching

Each case is described by a constructor call with free variables.

- Can think “expression with holes”

```plaintext
fun sum xs =
case xs of
  Nil => 0
  | Cons(y, ys) => y + (sum ys)
end
```

`Cons` case binds `y` to the head of the list and `ys` to the tail.

Scope of these variables is the body of the case.
Wildcards

Can use _ for a wildcard—matches but does not bind. _ can appear multiple times in a pattern

fun len xs =
case xs of
  Nil => 0
| Cons(_, xs) => 1 + (len xs)
end
Pattern matching

Matches can also be *deep*

data Tree a = Empty | Node (Tree a) a (Tree a)

fun rotate t =  
case t of  
    Node a x (Node (Node b y c)) z d  
    => Node (Node a x b) y (Node c z d)  
| _ => t  
end
Function definitions

Can fold pattern match into function definition

```
fun len Nil => 0
  | len Cons(_, xs) => 1 + (len xs)
```

VS.

```
fun len xs =
case xs of
  Nil => 0
  | Cons(_, xs) => 1 + (len xs)
end
```
Constructors need not be identifiers

```haskell
fun len [] = 0
  | len _::xs = 1 + (len xs)
```

`[]` is shorthand for `Nil`

`x::xs` is shorthand for `Cons(x, xs)`

`[x]` is shorthand for `x::[]`

`[x,y,z]` is shorthand for `x::y::z::[]`
Patterns + functions = concise code

// To sort array a[] of size n: qsort(a,0,n-1)
void qsort(int a[], int lo, int hi) {
    int h, l, p, t;
    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];
        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        } while (l < h);
        a[hi] = a[l];
        a[l] = p;
        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}

Haskell

qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
Matching in Scala

Use case classes to define algebraic data types
Match with the `match` expression
Can extend pattern matching with `extractors`
Case classes

sealed abstract class Tree

case class Empty extends Tree

case class Node(l: Tree, x: Int, r: Tree)

• sealed declares: these are the only subclasses

• ensures pattern matches are exhaustive
def sum(t: Tree) = t match {
  case Empty => 0
  case Node(l, x, r) => sum(l) + x + sum(r)
}
Destructuring

Can also use patterns on left-hand-side of an assignment

scala> var (x::xs) = List(1,2)
x: Int = 1
xs: List[Int] = List(2)
Desiderata

A pattern match should be complete, or exhaustive
• compiler should report when a case is missing

A pattern match should be unique
• compiler should report when cases overlap

Compiler can ensure this if patterns are just constructor calls + free variables
Side conditions

sealed abstract class List[A] {
  ...
  def filter(p: A => Boolean) = this match {
    case Nil => Nil
    case x::xs if p(x) => x::(xs.filter(p))
    case x::xs => xs.filter(p)
  }
}

Since cases can overlap, order matters
Red-black trees

Binary search tree where every node is labeled either red or black

Invariants

• no red node has a red child
• every path from a node to any leaf contains the same number of black nodes

Ensures longest path is no more than twice as long as shortest path from root ($B^k$ vs. $(BR)^k$)

Guaranteed $O(\log n)$ lookup, insertion, delete time
Scala red-black trees

class Color
  case object R extends Color
  case object B extends Color

class Tree
  case class E extends Tree
  case class T(c: Color, l: Tree, e: Int, r: Tree) extends Tree

def lookup(x: Int, t: Tree) = t match {
  case E => false
  case T(_, a, y, b) if x < y => lookup(x, a)
  case T(_, a, y, b) if x > y => lookup(x, b)
  case T(_, a, y, b) if x == y => true
  case T(_, a, y, b) if x == y => true
}
Scala red-black trees

def insert(x: Int, s: Tree) = {
    def ins(t: Tree) = t match {
        case E => T(R, E, x, E)
        case T(c, a, y, b) if x < y =>
            balance(c, ins a, y, b)
        case T(c, a, y, b) if x > y =>
            balance(c, a, y, ins b)
        case T(c, a, y, b) if x == y => t
    }
    T(_, a, y, b) = ins s
    T(B, a, y, b)
}
Scala red-black trees

```scala
def balance(c: Color, l: Tree, w: Int, r: Tree) = {
  (c, l, w, r) match {
    case (B, T(R, T(R, a, x, b), y, c), z, d) => T(R, T(B, a, x, b), y, T(B, c, y, d))
    case (B, T(R, a, x, T(R, b, y, c)), z, d) => T(R, T(B, a, x, b), y, T(B, c, y, d))
    case (B, a, x, T(R, T(R, b, y, c), z, d)) => T(R, T(B, a, x, b), y, T(B, c, y, d))
    case (B, a, x, T(R, b, y, T(R, c, z, d))) => T(R, T(B, a, x, b), y, T(B, c, y, d))
    case _ => T(c, l, w, r)
  }
}
```
Functional programming examples
List append

Scala:
• \( xs ::: ys \)

ML:
• \( xs @ ys \)

Scheme
• \( (\text{append } xs \; ys) \)
def append[A](xs: List[A], ys: List[A]): List[A] = xs match {
  case Nil => ys
  case z::zs => z::append(zs, ys)
}
List map

def map[A,B](f: A => B)(list: List[A]): List[B]
def map[A, B](f: A => B)(list: List[A]): List[B] =
    list match {
        case Nil => Nil
        case x::xs => f(x)::(map(f)(xs))
    }
def reverse[A](list: List[A]): List[A] = list match {
  case Nil => Nil
  case x::xs => xs ::: List(x)   // ::: is append operator
}

List reversal
def reverse[A](list: List[A]): List[A] = {
  def rev2[A](list: List[A], acc: List[A]): List[A] = list match {
    case Nil => acc
    case x::xs => rev2(xs, x::acc)
  }
  rev2(list, Nil)
}
List reversal

xs.reverse
List fold

fold left
(List(1,2,3)).foldLeft(_ + _)(0) = (((0 + 1) + 2) + 3) = 6

fold right
(List(1,2,3)).foldRight(_ + _)(0) = (1 + (2 + (3 + 0))) = 6
Fold left

```scala
def foldLeft[B](f: (B, A) => B)(z: B)(list: List[A]): B =
  list match {
    case Nil => z
    case x::xs => foldLeft(f)(f(x, z))(xs)
  }
```
Fold right

def foldRight[B](f: (A,B) => B)(z: B)(list: List[A]): B =
  list match {
    case Nil => z
    case x::xs => f(x, foldRight(f)(z)(xs))
  }
def flatten[A](list: List[List[A]]): List[A] = list match {
  case Nil => Nil
  case x::xs => x ::: flatten(xs)
}

val xs = List(List(1,2,3),List(4,5,6))
flatten(xs)
def flatten2[A](list: List[List[A]], acc: List[A]): List[A] =
    list match {
    case Nil => acc
    case x::xs => flatten2(xs, acc ::: x)
    }

def flatten[A](list: List[List[A]]): List[A] = flatten2[A](list, Nil)
def flatten[A](list: List[List[A]]): List[A] = {
    def flatten2[A](list: List[List[A]], acc: List[A]): List[A] =
        list match {
            case Nil => acc
            case x::xs => flatten2(xs, acc ::: x)
        }
    flatten2[A](list, Nil)
}
List flattening

val xs = List(List(1,2,3),List(4,5,6))

xs.foldLeft(List[Int]())(_ ::: _)  

--> List(1,2,3,4,5,6)
List flattening

val xs = List(List(1,2,3),List(4,5,6))

xs.flatten

--> List(1,2,3,4,5,6)
Questions?