CSE 341
Lecture 10

more about data types; nullable types; option
Ullman 6.2 - 6.3; 4.2.5 - 4.2.6

slides created by Marty Stepp
http://www.cs.washington.edu/341/

Creating new types of data

datatype ***name*** = ***value*** | ***value*** | ... | ***value***;

- a new type that contains only a fixed set of values
  - analogous to the enum in Java/C

- Examples:
  - ```
    datatype CardSuit = Clubs | Diamonds
                      | Hearts | Spades;
    ```
  - `datatype Color = Red | Green | Blue;`
  - `datatype Gender = Male | Female;`
    ```

Datatype / case exercise

- Define a method `haircutPrice` that accepts an age and gender as parameters and produces the price of a haircut for a person of that age/gender.
  - Kids' (under 10 yrs old) cuts are $10.00 for either gender.
  - For adults, male cuts are $18.25, female cuts are $36.50.

- Solution:

```
fun haircutPrice(age, gend) =
        if age < 10 then 10.00
        else case gend of Male   => 18.25
                        | Female => 36.50;
```

Type constructors

a ***TypeCtor*** is either: `name of typeExpression`

or: `value`

```
datatype name = TypeCtor | TypeCtor ...
              | TypeCtor;
```

- `datatypes` don't have to be just fixed values!
  - they can also be defined via "type constructors" that accept additional information
  - patterns can be matched against each type constructor

Type constructor example

```
(* Coffee : type, caffeinated?
   Wine   : label, year
   Beer   : brewery name
   Water  : needs no parameters *)
datatype Beverage =
    Water
|   Coffee of string * bool
|   Wine of string * int
|   Beer of string;
```

`- val myDrink = Wine("Franzia", 2009);`

val myDrink = Wine ("Franzia",2009) : Beverage

`- val yourDrink = Water;`

val yourDrink = Water : Beverage

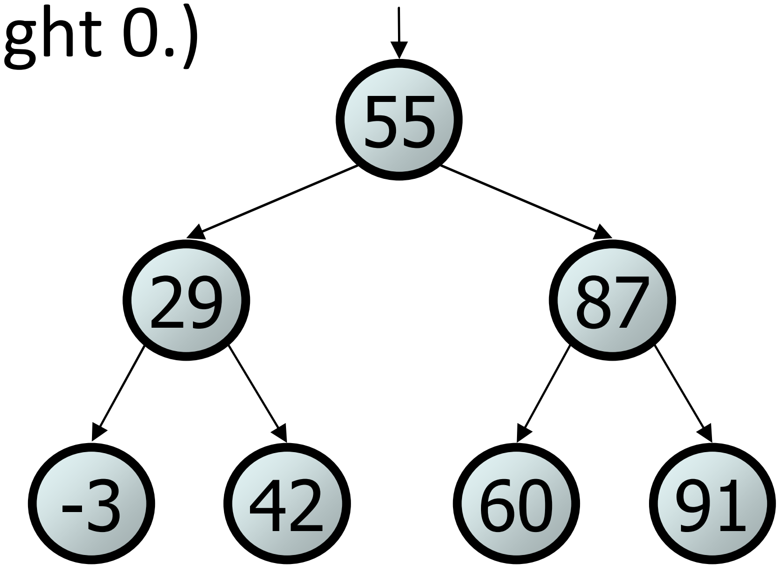
Patterns to match type ctors

```
(* Produces cafe's price for the given drink. *)
fun price(Water) = 1.50
|   price(Coffee(type, caf)) = if caf then 3.00
                               else 3.50
|   price(Wine(label, year)) = if year < 2009
                               then 30.0 else 10.0
|   price(Beer(_)) = 4.00;
```

- functions that process datatypes use patterns
  - pattern gives names to each part of the type constructor, so that you can examine each one and respond accordingly

Binary tree type exercise (6.3)

- Define a type `IntTree` for binary search trees of `int`s.
  - Define a function `add` that takes a tree and an integer and adds that value to the given tree in sorted order.
    - The function should produce the new tree as its result.
  - Define a function `height` to see how many levels are in a given tree. (Empty trees have height 0.)



Binary tree type solution

```
(* A type to represent binary search trees of integers. *)
datatype IntTree = Empty
                 | Node of int * IntTree * IntTree;

(* Adds the given value to the tree in order. *)
fun add(Empty, value) = Node(value, Empty, Empty)
|   add(n as Node(data, l, r), value) =
       if value < data then Node(data, add(l, value), r)
       else if value > data then Node(data, l, add(r, value))
       else n;

(* Produces the height of the given tree.  Empty is 0. *)
fun height(Empty) = 0
|   height(Node(_, left, right)) =
        1 + Int.max(height(left), height(right));
```

Concerning null

- **null**: A special empty value, often called "null" or "nil", that exists as part of the range of values of a type.
  - generally considered to be the absence of a value
  - many of the type's operations cannot be performed on null
  - What is the benefit of null? How is it used?
  - null was created by C.A.R. Hoare in 1965 as part of Algol W
    - Hoare later described null as a "billion dollar mistake"

How null is used (Java)

- `null` is often used to represent an error condition
  - `BufferedReader` returns `null` when input is done
  - `HashMap` returns `null` when `get` method cannot find key

- But this is done inconsistently...
  - `Scanner` throws an `IOException` when input is done
  - `ArrayList` returns -1 when `indexOf` cannot find a value
  - `System.in` returns -1 when it cannot read a character

- Not possible to return `null` for Java's primitive types

Java primitives and null

- In Java, object variables can be null; primitives cannot.
- Java's `int` type represents all integers: -2, -1, 0, 1, 2, 3, ...
  - How can we represent the lack (absence) of a number?
  - 0? -1? not appropriate because these are still legal integers
- Pretend that `int`s could be `null`. What would happen?

```
int noNumber = null;
System.out.println(noNumber);      // null
int x = noNumber + 4;              // exception
noNumber == null                   // true
noNumber == 2                      // false
noNumber > 5                       // exception? false?
noNumber <= 10                     // exception? false?
```

Other views of null

Some languages use alternatives to having a null value:

- **null object** pattern: Language provides an object that has predictable "empty" behavior.
  - can still call methods on it, but get back "empty" results
  - example: Difference in Java between `null` and `""`

- **option type** ("maybe type") pattern: Represents an optional value; e.g., a function that optionally returns.
  - A function can be declared to say, "I *might* return a value of type Foo, or I might return nothing at all."

Nullable types

- **nullable type**: A data type that contains null as part of its range of values.
  - In Java, every object type is nullable; primitives are not.

- In ML, only list types are nullable by default (`nil`, `[]`) .
  - but for *any* type, you can create a modified version of that type that *does* contain null (a nullable version of the type)
    - this is called an *option type*
    - example: `int option` is an `int` that can be null

Option types (4.2.5)

```
NONE           (* represents null *)
SOME expr      (* a value of a nullable type *)
```

- A function can be written to return an option type
  - some paths in the code return `NONE`
  - other paths return `SOME` *`value`*
    - analogy: a bit like an `Integer` wrapper over an `int` in Java
  - the calling code *must* explicitly specify how to deal with the "null case" (NONE) if it should occur, for it to compile

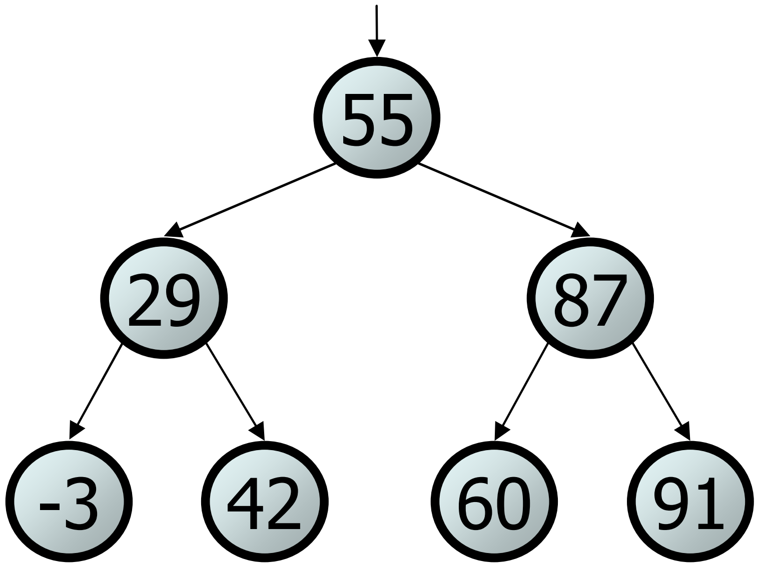
Playing with option types

```
- NONE;
```
val it = NONE : 'a option
```
- SOME;
```
val it = fn : 'a -> 'a option
```
- SOME 3;
```
val it = SOME 3 : int option
```
- SOME "hello";
```
val it = SOME "hello" : string option

- `isSome` ***x*** returns true if *x* is a SOME (not NONE)
- `valOf` ***x*** returns the value *v* stored in *x*, if *x* is SOME *v*
  - often not needed due to pattern matching (see next slide)

Option type exercise

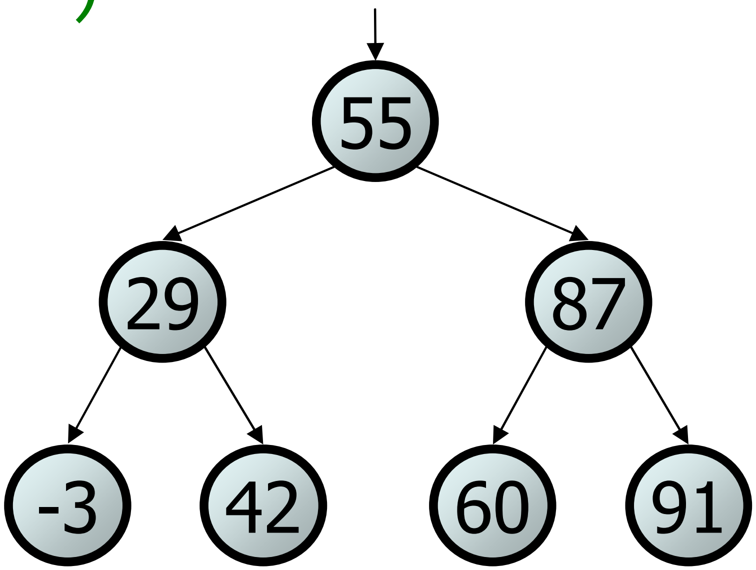
- Define a function `min` that produces the smallest integer value in a binary search tree of integers.
  - What if the tree is empty?



Option type solution

```
(* Produces the smallest value in the tree.
   Produces NONE if tree is empty. *)
fun min(Empty) = NONE
|   min(Node(data, left, right)) =
        if left = Empty then SOME data
        else min(left);

(* assuming IntTree t is defined *)
- min(t);
val it = SOME ~3 : int option
- valOf (min(t));
val it = ~3 : int
- min(Empty);
val it = NONE : int option
```



Option implementation and usage

- an option is just a simple `datatype` in ML:

```
datatype 'a option = NONE | SOME of 'a;
```

- most functions that use options use patterns for them:

```
case (min(t)) of
     NONE => "oops, empty"
|    SOME x => "min is " ^ Int.toString(x)
```

Option: the big picture

- Why not just throw an exception on an empty tree?

```
exception NoSuchElement;
fun min(Empty) = raise NoSuchElement
|   min(Node(data, left, right)) =
        if left = Empty then data
        else min(left);
```

  - either way is acceptable
    - the **exception** way allows "non-local" error handling
    - the **option** way forces the caller to think about null (NONE) and to explicitly handle the null case
  - Options allow carefully limited introduction of null into a program without forcing you to test for null everywhere.