Datatypes

The sky's the limit

15-150 M21

Lecture 0614
14 June 2021

Today's slogan:

If you can dream it, you can build it.

If you can build it, you can induct on it.

0 Trees in SML

Binary trees in SML

- We define a new type `tree` with the following syntax:

0614.0 (treeDefn.sml)

```
datatype tree =
  Empty | Node of tree * int * tree
```

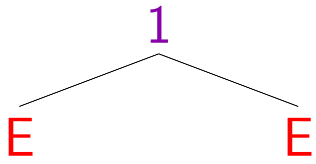
- This declares a new type called `tree` whose constructors are `Empty` and `Node`. `Empty` is a *constant constructor* because it's just a value of type `tree`. `Node` takes in an argument of type `tree*int*tree` and produces another `tree`.
- All trees are either of the form `Empty` or `Node(L,x,R)` for some `x : int` (referred to as the *root* of the tree), some `L : tree` (referred to as the *left subtree*), and some `R : tree` (referred to as the *right subtree*)

Arboretum

E

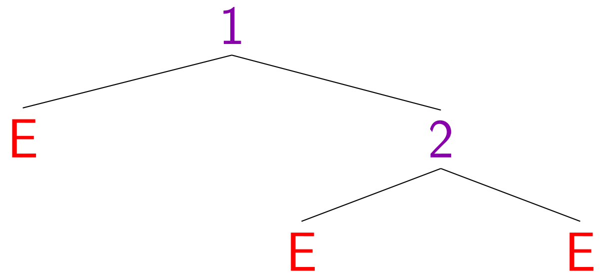
0614.6 (arboretum.sml)

```
val T0 = Empty
```



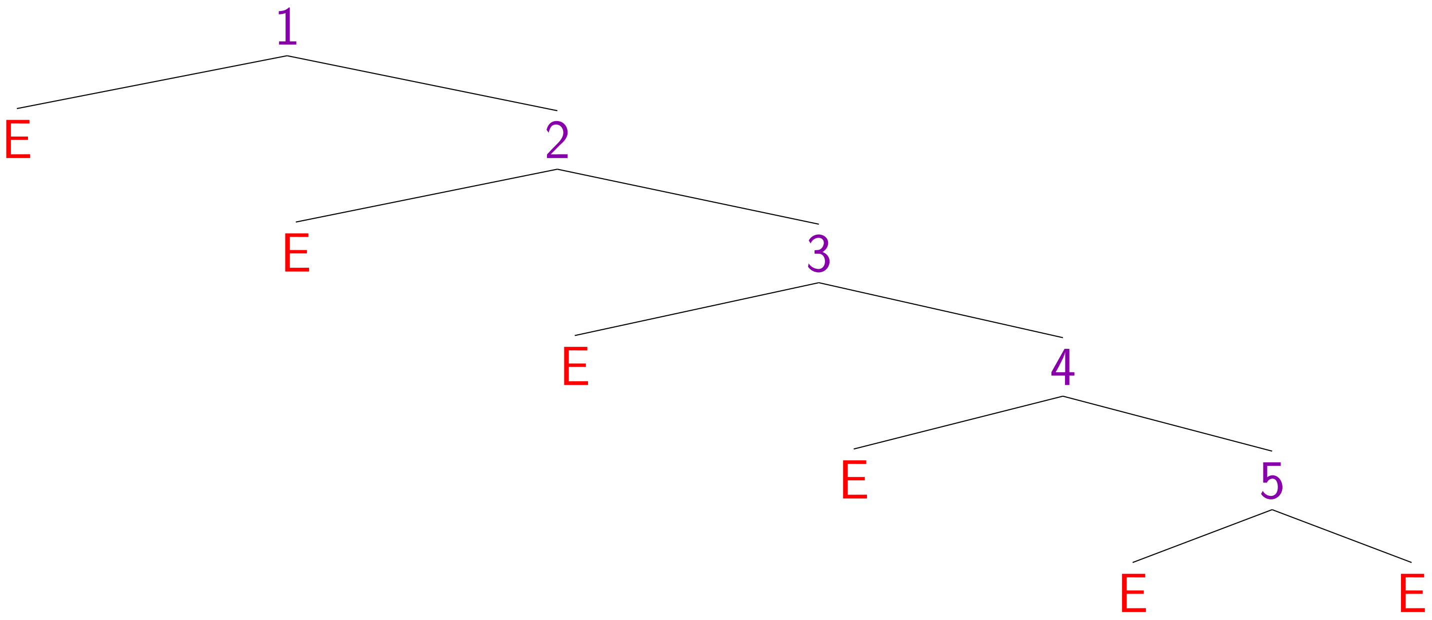
0614.7 (arboretum.sml)

```
val T1 = Node(Empty,1,Empty)
```



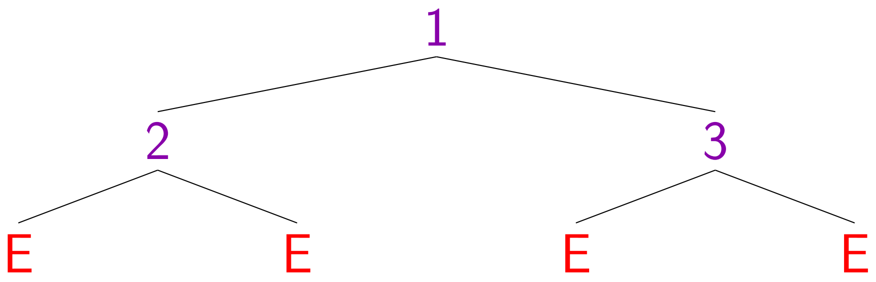
0614.8 (arboretum.sml)

```
val T2 = Node(Empty,1,Node(Empty,2,Empty))
```



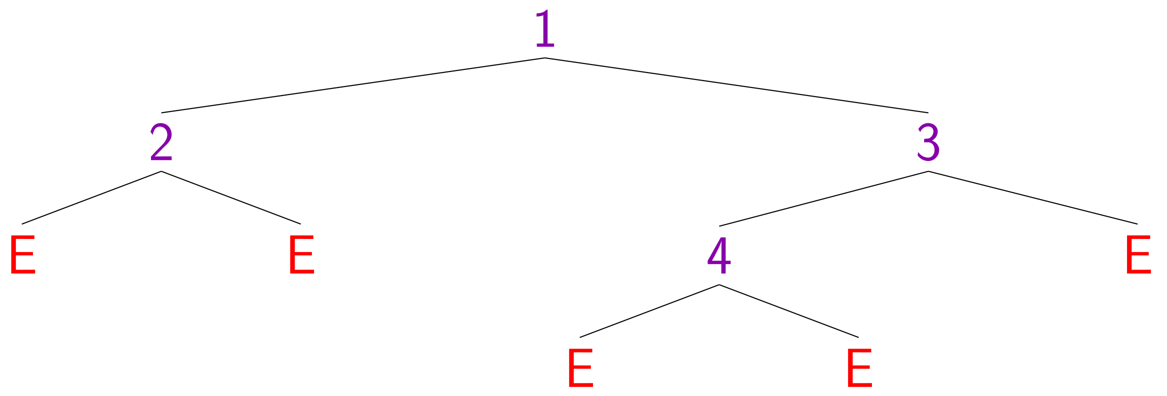
0614.9 (arboretum.sml)

```
val T3 = Node(Empty,1,Node(Empty,2,Node(Empty
  ,3,Node(Empty,4,Node(Empty,5,Empty)))))
```



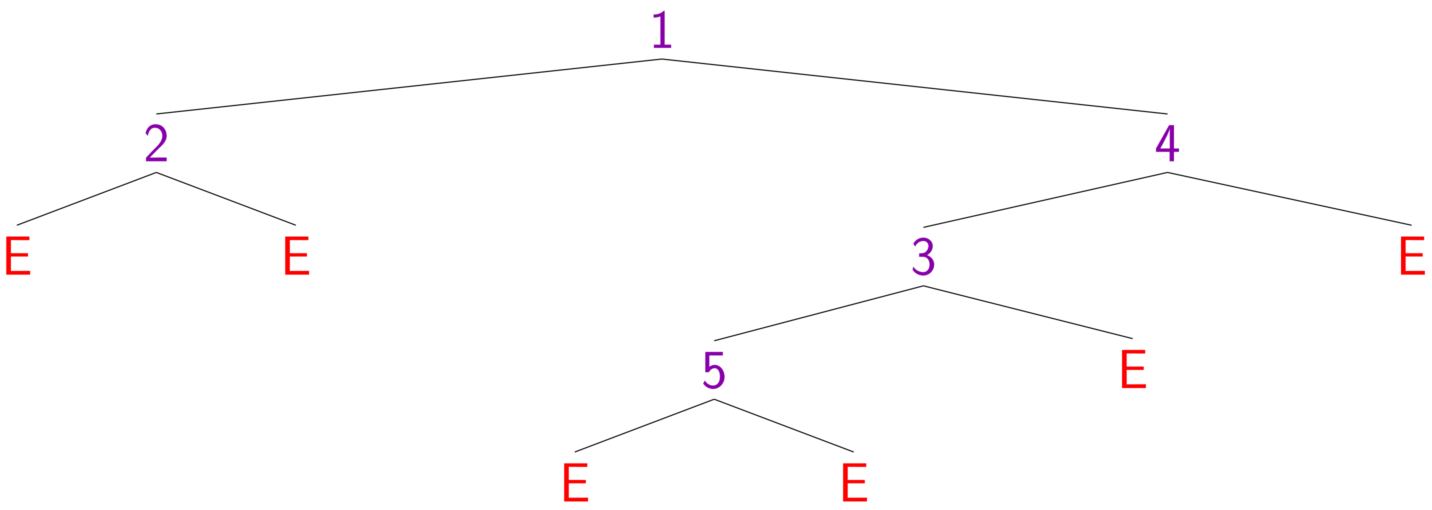
0614.10 (arboretum.sml)

```
val T4 = Node(Node(Empty,2,Empty),1,Node(Empty
  ,3,Empty))
```



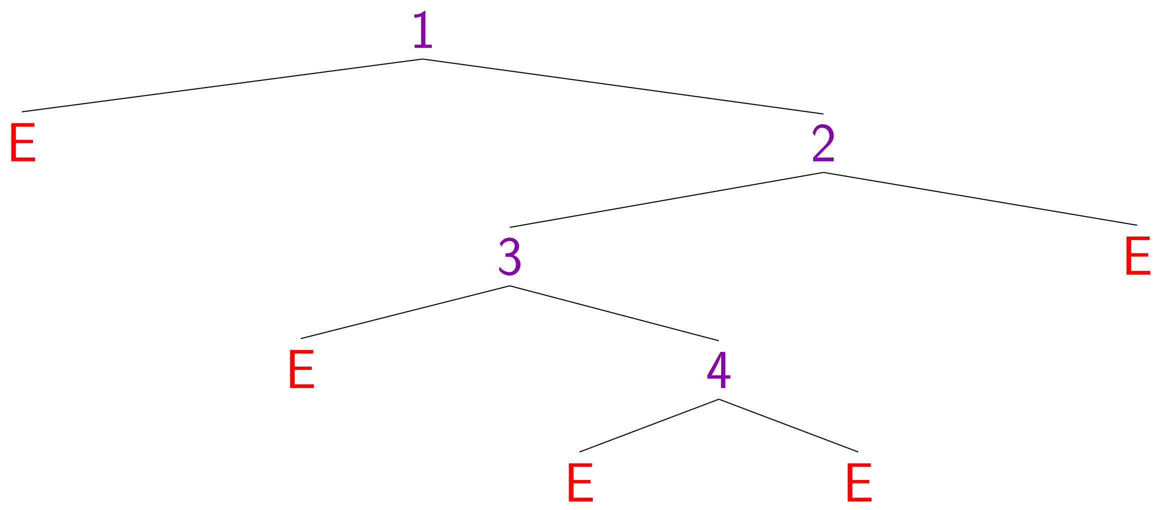
0614.11 (arboretum.sml)

```
val T5 = Node(Node(Empty,2,Empty),1,Node(Node(
  Empty,4,Empty),3,Empty))
```



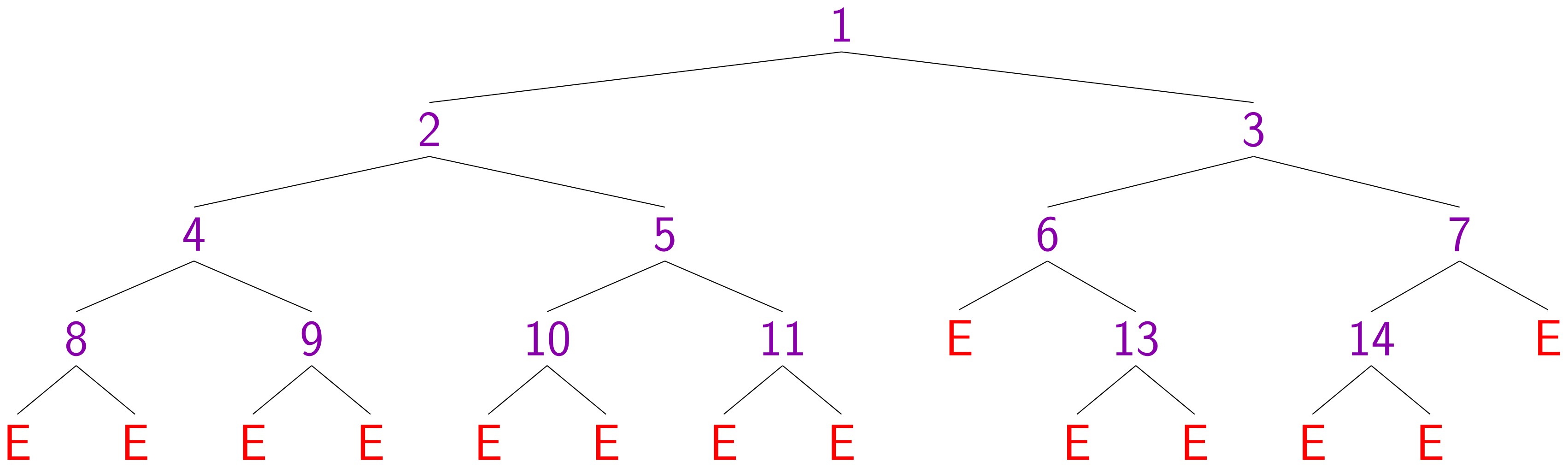
0614.12 (arboretum.sml)

```
val T6 = Node(Node(Empty,2,Empty),1,Node(Node(
  Node(Empty,5,Empty),3,Empty),4,Empty))
```



0614.13 (arboretum.sml)

```
val T7 = Node(Empty,1,Node(Node(Empty,3,Node(
  Empty,4,Empty)),2,Empty))
```



0614.14 (arboretum.sml)

```
val T8 = Node(Node(Node(Node(Empty,8,Empty),4,
  Node(Empty,9,Empty)),2,Node(Node(Empty,10,
  Empty),5,Node(Empty,11,Empty))),1,Node(Node(
  Empty,6,Node(Empty,13,Empty)),3,Node(Node(
  Empty,14,Empty),7,Empty)))
```

Height (or *depth*):

0614.1 (trees.sml)

```
fun height (Empty:tree):int = 0
  | height (Node(L,_,R)) =
      1 + Int.max(height L,height R)
```

Size

0614.2 (trees.sml)

```
fun size (Empty:tree):int = 0
  | size (Node(L,_,R)) =
      1 + size L + size R
```

Live Coding: Traversal

0614.3 (trees.sml)

```
fun inord (Empty:tree):int list = []
  | inord (Node(L,x,R)) =
     (inord L) @ (x::inord R)

fun preord (Empty:tree):int list = []
  | preord (Node(L,x,R)) =
     x::((preord L) @ (preord R))
```

When analyzing tree function, we have *two* standard notions of size:

- Depth/height, d
- Size (number of nodes), n

To simplify our analysis, we often assume the tree in question is **balanced**. A tree `Node(L,x,R)` is balanced iff

- `L` and `R` have approximately the same number of nodes
- Both `L` and `R` are balanced

A balanced tree of depth d will have approximately 2^d nodes

Demonstration: `treesum`

runtime analysis

0614.4 (trees.sml)

```
fun treesum (Empty:tree):int = 0
  | treesum (Node(L,x,R)) =
        x+(treesum L)+(treesum R)
```

Depth-Analysis of `treesum`

0 Notion of size: depth d of the input

1 Recurrences:

$$W(0) = k$$
$$W(d) = 2W(d-1) + k$$

NOTE: This assumes the tree is balanced

$$S(0) = k$$
$$S(d) = S(d-1) + k$$

2-4 ...

5 $W(d)$ is $O(2^d)$, $S(d)$ is $O(d)$

Demonstration: `find` runtime analysis

0614.5 (trees.sml)

```
fun find (_:int,Empty:tree):bool = false
  | find (y,Node(L,x,R)) =
        x=y orelse
        let
            val (resL,resR) =
                (find(y,L),find(y,R))
        in
          resL orelse resR
        end
```

Check Your Understanding

Why not `find(y,L) orelse find(y,R)`?

0 Notion of size: number of nodes n of the input

1 Recurrences:

$$W(0) = k$$
$$W(n) = 2W(n/2) + k$$

NOTE: This assumes the tree is balanced

$$S(0) = k$$
$$S(n) = S(n/2) + k$$

2-4 ...

5 $W(n)$ is $O(n)$, $S(n)$ is $O(\log n)$

Compare

```
fun find'(y,[]) = false
  | find'(y,x::xs) = x=y orelse find(y,xs)
```

This also has $O(n)$ work, but its span is $O(n)$ because there's no opportunity for parallelism!

Recall that for lists, the two constructors were `[]` and `:: of t * t list` where `t` is the type of list we're dealing with.
Subsequently, the induction principle for lists was that if P(`[]`) and if P(`xs`) implies P(`x::xs`), then P(`L`) holds for all `L`.

Principle of Structural Induction on Trees:
If

- P(`Empty`) holds
- for all values `L:tree`, `R:tree` and values `x:int`

P(`L`) and P(`R`) implies P(`Node(L,x,R)`)

then *for all values* `T : tree`, P(`T`) holds.

Example: Reversing Trees

0614.15 (moretrees.sml)

```
fun revTree (Empty : tree):tree = Empty
  | revTree (Node (L,x,R) =
        Node(revTree R,x,revTree L)
```

0614.3 (trees.sml)

```
fun inord (Empty:tree):int list = []
  | inord (Node(L,x,R)) =
      (inord L) @ (x::inord R)
```

Thm. For all values `T:tree`,

$$\texttt{rev (inord T)} \cong \texttt{inord(revTree T)}$$

When life hands you lemmas...

Lemma 1 For all valuable expressions `L1 : int list`, `L2 : int list`,

$$\texttt{rev (L1@L2)} \cong \texttt{(rev L2)@(rev L1)}$$

Lemma 2 `inord` is total

Lemma 3 `rev` is total

Lemma 4 For all valuable expressions `L1 : int list`, `L2 : int list`, and all values `x : int`,

$$\texttt{(L1@[x])@L2} \cong \texttt{L1@(x::L2)}$$

Lemma 5 `revTree` is total

Thm. For all values `T : tree`,

$$\texttt{rev (inord T)} \cong \texttt{inord(revTree T)}$$

Proof.

BC `T=Empty`

`rev (inord Empty)`	
$\cong$ `rev []`	(defn of `inord`)
$\cong$ `[]`	(defn of `rev`)
$\cong$ `inord Empty`	(defn `inord`)
$\cong$ `inord (revTree Empty)`	(defn `revTree`)

Example: Reversing Trees

IS `T=Node(L,x,R)` for some values `L,R:tree` and `x:int`

IH1 `rev(inord L)` $\cong$ `inord(revTree L)`

IH2 `rev(inord R)` $\cong$ `inord(revTree R)`

```
rev(inord (Node(L,x,R)))
 ≅ rev((inord L)@(x::(inord R)))                 (defn inord)
 ≅ (rev (x::inord R)) @ (rev(inord L))           Lemmas 1,2
 ≅ ((rev (inord R))@[x]) @ (rev(inord L))
                                     ( Lemma 2 , defn of rev)
 ≅ (rev (inord R))@(x::(rev(inord L)))           Lemmas 2,3,4
```

$\cong$ `(rev (inord R))@(x::(rev(inord L)))` Lemmas 2,3,4

$\cong$ `inord(revTree R) @ (x::inord(revTree L))` IH 1,2

$\cong$ `inord(Node(revTree R,x,revTree L))`

(Lemma 5, defn `inord`)

$\cong$ `inord(revTree(Node(L,x,R)))` (defn `revTree`)

$\square$

5-minute break

1 That's My Type

Notice some similarities...

- All natural numbers are either `0` or `n+1` for some natural number `n`. To prove P(`n`) for all natural numbers `n`, we prove P(`0`) and prove that P(`n`) implies P(`n+1`).
- All values of type `t list` are either `[]` or `x::xs` for some `x:t` and some value `xs:t list`. To prove P(`L`) for all values `L:int list`, we prove P(`[]`) and prove that P(`xs`) implies P(`x::xs`) for arbitrary `x:t`.
- All value of type `tree` are either `Empty` or `Node(L,x,R)` for some `x:int` and some values `L` and `R` of type `tree`. To prove P(`T`) for all values `T:tree`, we prove P(`Empty`) and prove that P(`L`) and P(`R`) together imply P(`Node(L,x,R)`) for arbitrary `x:int`.
- What's the general pattern?

0614.16 (datatypes.sml)

```
datatype foo = Abcd
             | Qwerty of int * string
             | Zyxwv of int * foo
```

- `Abcd` is a *constant constructor*, i.e. a constructor value of type `foo`
- `Qwerty` is a constructor of the `foo` type, which takes in an argument of type `int*string`. `Qwerty` can also be thought of (and used) as a function value of type `int * string -> foo`.
- `Zyxwv` is a constructor of the `foo` type, which takes in an argument of type `int * foo`. `Zyxwv` can also be thought of (and used) as a function value of type `int * foo -> foo`

0614.17 (datatypes.sml)

```
val f1 : foo = Abcd
val f2 : foo = Qwerty (15, "onefifty")
val f3 : foo = Zyxwv (150, f2)
```

0614.18 (datatypes.sml)

```
fun toInt Abcd = 2
  | toInt (Qwerty(n,_)) = n
  | toInt (Zyxwv(k,F)) = k + toInt F
```

Induction on defined datatypes

Thm. For all values `f : foo`, P(`f`).

Proof by induction on `f`

BC `f=Abcd`

(proof of P(`Abcd`))

BC `f=Qwerty(n,s)` for some values `n:int`, `s:string`

(proof of P(`Qwerty(n,s)`) for arbitrary `n,s`)

IS `f=Zyxwv(n,f')` for some values `n:int`, `f':foo`

IH P(`f'`)

(proof of P(`Zyxwv(n,f')`) for arbitrary `n`, using **IH**)

□

Demonstration: Pretty-printed `nats`

```
datatype nat = Zero | Succ of nat

fun toInt Zero = 0
  | toInt (Succ N) = 1+(toInt N)
(* REQUIRES: n>=0 *)
fun nat 0 = Zero
  | nat n = Succ(nat (n-1))
fun toString N =
  Int.toString (toInt N)
infix ++
fun Zero ++ M = M
  | (Succ N) ++ M = Succ(N ++ M)
```

Zero, One, Two, Three

- ```
  datatype void = Void of void
  ```
- (built-in)
  ```
  datatype unit = ()
  ```
- (built-in)
  ```
  datatype bool = true | false
  ```
- (built-in)
  ```
  datatype order = LESS | EQUAL | GREATER
  ```
  ```

Module: Timing

github.com/smlhelp/aux-library/blob/main/Timing.sml aux-library/Timing.sml

```
type day = int
datatype month = Jan | Feb | Mar | Apr
               | May | Jun | Jul | Aug
               | Sep | Oct | Nov | Dec
type year = int
type date = year * month * day
```

Negative values of `YY` are interpreted as BCE, e.g.

```
val idesOfMarch = (~44,Mar,15)
```

Invariant: For any value `(YY,MM,DD) : date`, the value `YY` is not `0`: the year after 1 BCE (`~1`) was 1 CE (`1`), so there is no `0`.

date decoding functions

aux-library/Timing.sml

```
val year : date -> year
val month : date -> month
val day : date -> day
```

Note: it's possible to use the same name for the *type* `year` and the *value* `year`, since types and values have distinct namespaces. Make good choices.

```
val year = fn (YY,_,_) => YY
val month = fn (_,MM,_) => MM
val day = fn (_,_,DD) => DD
```

Leap Years

Leap Year Rules: How to Calculate Leap Years

In the Gregorian calendar, three criteria must be taken into account to identify leap years:

- **The year must be evenly divisible by 4;**
- If the year can also be evenly divided by 100, **it is *not* a leap year;**

unless...

- The year is also evenly divisible by 400. **Then it *is* a leap year.**

According to these rules, the years 2000 and 2400 are leap years, while 1800, 1900, 2100, 2200, 2300, and 2500 are *not* leap years.

aux-library/Timing.sml

```
fun leapYear YY =
((YY mod 4 = 0) andalso (YY mod 100 <> 0))
    orelse
(YY mod 400 = 0)
```

Number-of-Days Invariant

aux-library/Timing.sml

```
val numDays : month * year -> int
```

aux-library/Timing.sml

```
fun numDays (MM,YY) =
    case MM of
       Sep => 30
     | Apr => 30 | Jun => 30 | Nov => 30

     | Jan => 31 | Mar => 31 | May => 31
     | Jul => 31 | Aug => 31 | Oct => 31
     | Dec => 31

                 YY)
```

Invariant: For any value `(YY,MM,DD) : date`,

$$0 < \texttt{DD} \leq \texttt{numDays(MM,YY)}$$

The `Timing` module has its own custom exception, `Invalid`.

aux-library/Timing.sml

```
fun date (YY,MM,DD):date =
  let
    val _ = (YY <> 0) orelse raise Invalid
    val _ = (
              (0 < DD) andalso
              (DD <= (numDays (MM,YY)))
            )
            orelse raise Invalid
  in
    (YY,MM,DD)
```

```
fun monthSucc MM =
  case MM of
      Jan => Feb | Feb => Mar | Mar => Apr
    | Apr => May | May => Jun | Jun => Jul
    | Jul => Aug | Aug => Sep | Sep => Oct
    | Oct => Nov | Nov => Dec | Dec => Jan
fun monthPred MM =
  case MM of
      Jan => Dec | Feb => Jan | Mar => Feb
    | Apr => Mar | May => Apr | Jun => May
    | Jul => Jun | Aug => Jul | Sep => Aug
    | Oct => Sep | Nov => Oct | Dec => Nov
```

Self-Documenting Code

Slick Pattern-Matching

aux-library/Timing.sml

```
(* datePred : date -> date *)
fun datePred (YY,Jan,1) =
        (yearPred YY,Dec,31)
  | datePred (YY,MM,1) =
        (YY, monthPred MM,
         numDays(monthPred MM,YY))
  | datePred (YY,MM,DD) = (YY,MM,DD-1)
(* dateSucc : date -> date *)
fun dateSucc (YY,Dec,31) =
        (yearSucc YY,Jan,1)
  | dateSucc (YY,MM,DD) =
        if DD = (numDays (MM,YY))
        then (YY,monthSucc MM,1)
        else (YY,MM,DD+1)
```

More examples from Timing

```
datatype weekday = Sunday | Monday | Tuesday |
  Wednesday | Thursday | Friday | Saturday
fun weekdaySucc W = case W of
      Sunday => Monday
    | Monday => Tuesday
```

```
val dateToString : date -> string
```

```
type timezone
```

```
val Local : timezone
```

```
val dayOfWeek : timezone -> weekday
```

```
val today : timezone -> date
```

- We can write recursive functions operating on trees, analyze those functions asymptotically using the tree method, and prove properties about them by structural induction
- The SML `datatype` keyword allows us to declare our own custom datatypes, to better encode data
- Given any recursive datatype, we can determine a recursion principle and a principle of structural induction

Next Time

- Parametrizing datatypes by type variables
- Polymorphism
- Polymorphic Sort

Thank you!