Lecture 7
Principles of Functional Programming
Summer 2020

Datatypes

Section 1

Trees

Binary trees in SML

- We define a new type `tree` with the following syntax (which we'll discuss more later):

7.0

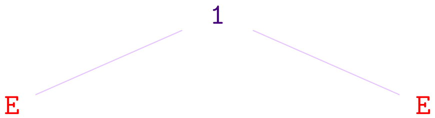
```
datatype tree =
  Empty | Node of tree * int * tree
```

- 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*)

E

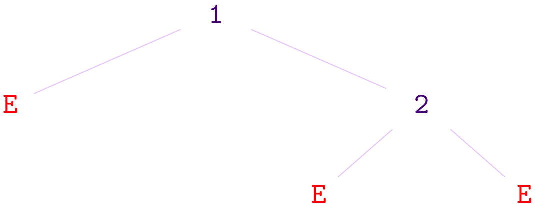
7.2

```
Empty
```



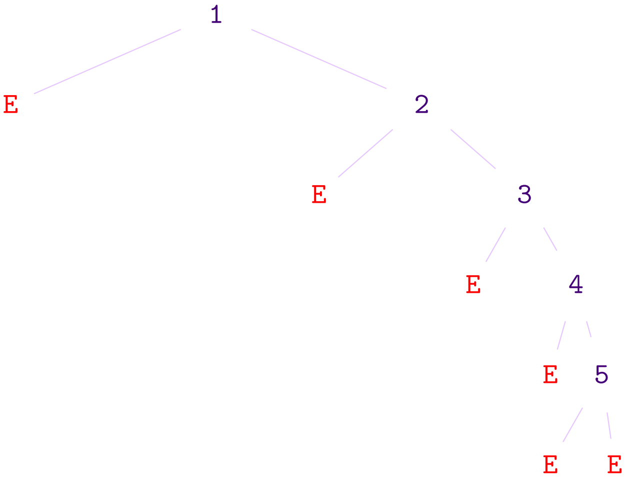
7.3

```
Node(Empty,1,Empty)
```



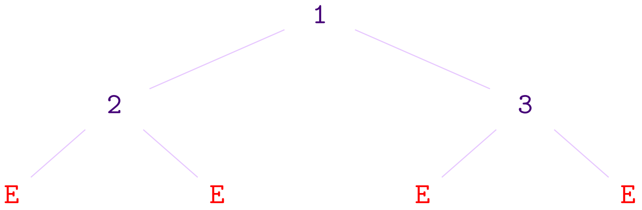
7.4

```
Node(Empty,1,Node(Empty,2,Empty))
```



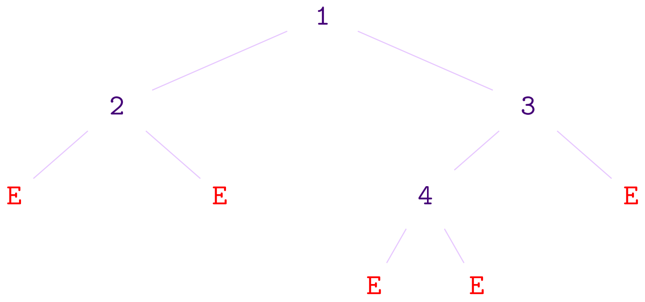
7.5

```
Node(Empty,1,Node(Empty,2,Node(Empty,3,Node(
    Empty,4,Node(Empty,5,Empty)))))
```



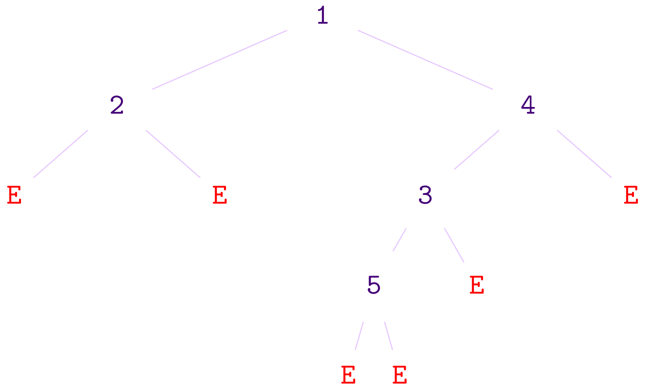
7.6

```
Node(Node(Empty,2,Empty),1,Node(Empty,3,Empty))
```



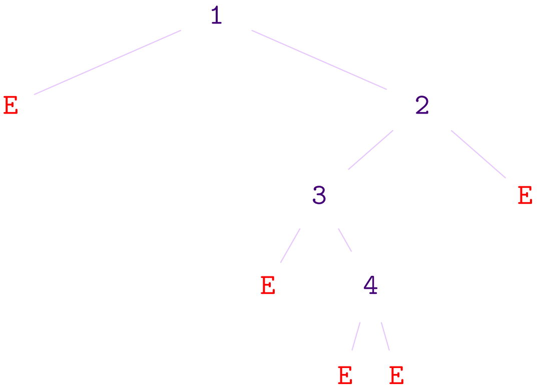
7.7

```
Node(Node(Empty,2,Empty),1,Node(Node(Empty,4,
    Empty),3,Empty))
```



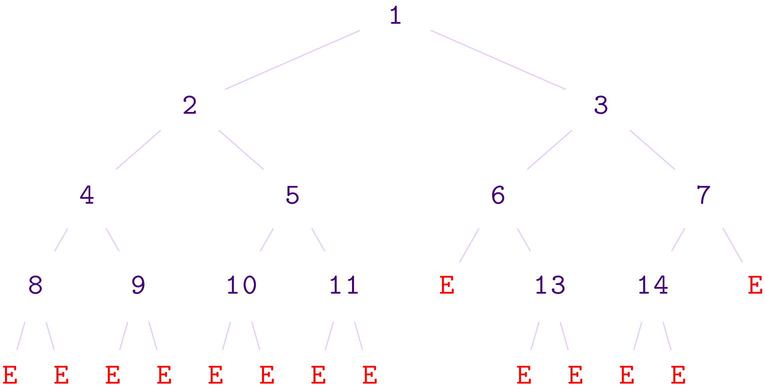
7.8

```
Node(Node(Empty,2,Empty),1,Node(Node(Node(
    Empty,5,Empty),3,Empty),4,Empty))
```



7.9

```
Node(Empty,1,Node(Node(Empty,3,Node(Empty,4,Empty)),2,Empty))
```



7.10

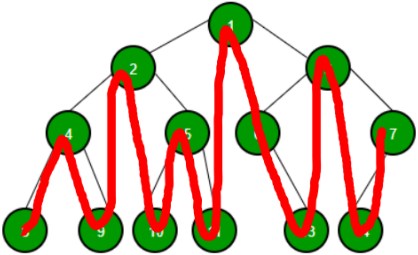
```
Node(Node(Node(Node(Empty,8,Empty),4,Node(
    Empty,9,Empty)),2,Node(Node(Empty,10,Empty)
    ,5,Node(Empty,11,Empty))),1,Node(Node(Empty
    ,6,Node(Empty,13,Empty)),3,Node(Node(Empty
    ,14,Empty),7,Empty)))
```

Traversals

Inorder

7.12

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

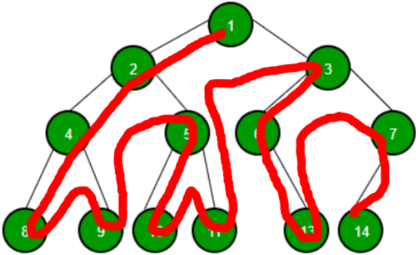


Traversals

Preorder

7.13

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



Minimum

7.14

```
fun min (Empty:tree, default:int) = default
  | min (Node(L,x,R),default) =
      let
        (* Parallel *)
        val (minL,minR) =
            (min(L,default), min(R,default))
      in
        (* Constant-time *)
        Int.min(x,Int.min(minL,minR))
      end
```

Analyzing the work & span of tree functions

To analyze the runtime complexity of functions defined by recursion on trees, we need a notion of size for trees. It turns out that we have *two*:

- Depth/height: the length (number of nodes) in the longest path from the root to any leaf node

7.1

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

- Size: the number of nodes in the tree

7.11

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

We'll use both.

Balanced Trees

We'll say a tree is *balanced* if both its subtrees are balanced and both of its subtrees have approximately the same height (their heights differ by at most one).

- On balanced trees, you can assume a recursive call to the left subtree costs approximately the same amount of time as on the right subtree.
- If n is the size of a balanced tree, and d is its height, then we can assume

$$n \approx 2^d$$

Depth-Analysis of `min`

0 Notion of size: depth d of the input tree

1 Recurrences:

$$W_{\texttt{min}}(0) = k_0$$
$$W_{\texttt{min}}(d) \leq k_1 + 2W_{\texttt{min}}(d-1)$$

$$S_{\texttt{min}}(0) = k_0$$
$$S_{\texttt{min}}(d) \leq k_1 + S_{\texttt{min}}(d-1)$$

2-4 ...

5 $W_{\texttt{min}}(d)$ is $O(2^d)$, $S_{\texttt{min}}(d)$ is $O(d)$

Remember: if the input tree is balanced, then $2^d \approx n$, where n is the size (number of nodes).

Size-Analysis of preord

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

1 Recurrences:

$$W_{\texttt{preord}}(0) = k_0$$
$$W_{\texttt{preord}}(n) = 2W_{\texttt{preord}}(n/2) + kn$$

NOTE: This assumes the tree is balanced

$$S_{\texttt{preord}}(0) = k_0$$
$$S_{\texttt{preord}}(n) \leq S_{\texttt{preord}}(n/2) + kn$$

2-4 ...

5 $W_{\texttt{preord}}(n)$ is $O(n \log n)$, $S_{\texttt{preord}}(n)$ is $O(n)$

(pause for questions)

Section 2

Structural Induction

Induction Principle

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 $P(\texttt{[]})$ and if $P(\texttt{xs})$ implies $P(\texttt{x::xs})$, then $P(\texttt{L})$ holds for all `L`.

Principle of Structural Induction on Trees: If $P(\texttt{Empty})$ holds and, for all values `L:tree`, `R:tree` and values `x:int`, $P(\texttt{L})$ and $P(\texttt{R})$ implies $P(\texttt{Node(L,x,R)})$.

Example: Reversing Trees

7.15

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

7.12

```
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 of Thm

BC `T=Empty`

$$
\begin{aligned}
&\texttt{rev (inord Empty)} \\
&\cong \texttt{rev []} && \text{(defn of \texttt{inord})} \\
&\cong \texttt{[]} && \text{(defn of \texttt{rev})} \\
&\cong \texttt{inord Empty} && \text{(defn \texttt{inord})} \\
&\cong \texttt{inord (revTree Empty)} && \text{(defn \texttt{revTree})}
\end{aligned}
$$

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)))
```

$\cong$ `rev((inord L)@(x::(inord R)))` (defn `inord`)

$\cong$ `(rev (x::inord R)) @ (rev(inord L))` (Lemma 1,2)

$\cong$ `((rev (inord R))@[x]) @ (rev(inord L))`
(Lemma 2, defn of `rev`)

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

Example: Reversing Trees

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

$\cong$ `inord(revTree R) @ (x::inord(revTree L))` (IH1,2)

$\cong$ `inord(Node(revTree R,x,revTree L))` (Lemma 5, defn `inord`)

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

$\square$

(pause for questions)

Section 3

Datatypes

Notice some similarities...

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

The datatype keyword

7.16

```
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`

Recursion on defined datatypes

7.17

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

7.18

```
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**)

$\square$

Examples

- Natural Numbers

```
datatype nat = Zero
             | Succ of nat

fun toInt Zero = 0
  | toInt (Succ n) = 1 + toInt n
fun fromInt 0 = Zero
  | fromInt n = Succ(fromInt (n-1))
```

Note: `natFact` is total, even though `fact` is not:

```
fun fact 0 = 1 | fact n = n * fact(n-1)

fun natFact (N : nat):int =
    fact(toInt N)
```

Examples

- Trees

7.0

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

- Lists

```
datatype 't list =
     [] | :: of 't * 't list
infixr ::
```

(Note: This is not *exactly* how lists are defined)

  - Parametrized by a type variable (more about this on Monday)
  - `::` is also infixed

New Example: `options`

The parametrized datatype `option` is pre-defined in SML:

```
datatype 't option = NONE | SOME of 't
```

- For every type `t`, there is a type `t option`
- `NONE` is a value (and a constructor) of type `t option`.
- `SOME` is a constructor of the `t option` type: if `x:t`, then `SOME(x)` is a value of type `t option`. `SOME` is also a function value of type `t -> t option`.
- We can case on options by pattern-matching the constructors:

```
case (thing : bool option option) of
  (SOME(SOME true)) => ...
| (SOME _ ) => ...
| NONE => ...
```

- Can do structural induction on options

Section 4

Example: Days of the Week

Type Aliases

7.19

```
type giantTuple = int * int * int * int * int
    * int * int * int
```

Days of the Week

7.20

```
datatype day =
    Sunday | Monday | Tuesday | Wednesday |
   Thursday | Friday | Saturday
```

7.21

```
fun nextDay Saturday = Sunday
  | nextDay Friday = Saturday
  | nextDay Thursday = Friday
  | nextDay Wednesday = Thursday
  | nextDay Tuesday = Wednesday
  | nextDay Monday = Tuesday
  | nextDay Sunday = Monday
```

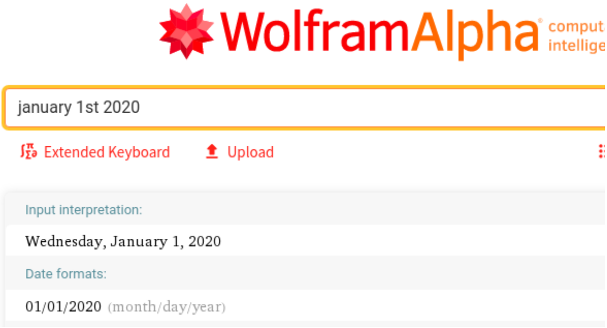
Day of the Week

7.22

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

`dayOfWeek : month * date -> day`
REQUIRES: DD ≥ 0
ENSURES: `dayOfWeek(MM,DD)` evaluates to what day of the week it was on the DDth day of the month of MM, 2020. Each month is counted as if it went on forever, so `dayOfWeek(Apr,197000)` should return what day of the week it is, 196970 days after April 2020 concludes.

Hard-code New Year's Day



7.23

```
fun dayOfWeek (Jan:month,01:date):day =
    Wednesday
```

Carry over months

7.24

```
    | dayOfWeek (Feb,01) =
          nextDay(dayOfWeek (Jan,31))
    | dayOfWeek (Mar,01) =
          nextDay(dayOfWeek (Feb,29))
    | dayOfWeek (Apr,01) =
          nextDay(dayOfWeek (Mar,31))
    | dayOfWeek (May,01) =
          nextDay(dayOfWeek (Apr,30))
    | dayOfWeek (Jun,01) =
          nextDay(dayOfWeek (May,31))
    | dayOfWeek (Jul,01) =
          nextDay(dayOfWeek (Jun,30))
    | dayOfWeek (Aug,01) =
          nextDay(dayOfWeek (Jul,31))
    | dayOfWeek (Sep,01) =
          nextDay(dayOfWeek (Aug,31))
```

Then recur

7.25

```
  | dayOfWeek (MM,DD) = nextDay(dayOfWeek (MM,
  DD-1))
```

Thank you!