Overview

Learning objectives:
- Show how recursive datatypes can be used to represent trees.
  Examples:
  - search trees
  - expression trees
- Introduce abstract datatypes.

Trees

A finite tree is a value which may contain a subcomponent of the same type.

Example: A binary search tree

Condition: for every node containing the value $x$: every value in the left subtree is smaller than $x$, and every value in the right subtree is greater than $x$.

Binary Trees

A recursive datatype is used to represent values which are trees.

datatype tree = Lf | Br of tree*int*tree;
> datatype tree
> con Lf = Lf : tree
> con Br = fn : tree * int * tree -> tree
Binary search trees: Insertion

Recursion on the structure of trees:

- Constructors `Lf` and `Br` are used in patterns

```haskell
fun insert (i, Lf) = Br(Lf, i, Lf)
| insert (i, tr as Br(t1, j, t2)) =
  case Int.compare(i, j) of
    EQUAL => tr
    | LESS => Br(insert(i, t1), j, t2)
    | GREATER => Br(t1, j, insert(i, t2))
```

- The search tree condition is an invariant for `insert`

Example:
- `val t1 = Br(Lf, 3, Br(Lf, 5, Lf));`
- `val t2 = insert(4, t1);`
- `val t2 = Br(Lf, 3, Br(Br(Lf, 4, Lf), 5, Lf)) : tree`

Binary search trees: `member` and `toList`

```haskell
fun member (i, Lf) = false
| member (i, Br(t1, j, t2)) =
  case Int.compare(i, j) of
    EQUAL => true
    | LESS => member(i, t1)
    | GREATER => member(i, t2)
```

In-order traversal

```haskell
fun toList Lf = []
| toList (Br(t1, j, t2)) = toList t1 @ [j] @ toList t2;
```

```
> val toList = fn : tree -> int list
```

Deletions in search trees

Delete minimal element in a search tree: `tree -> int * tree`

```haskell
fun delMin (Br(Lf, i, t2)) = (i, t2)
| delMin (Br(t1, i, t2)) = let val (m, t1’) = delMin t1
  in (m, Br(t1’, i, t2)) end
```

Delete element in a search tree: `tree -> int -> tree`

```haskell
fun delete (Lf, i) = Lf
| delete (Br(t1, i, t2), j) =
  case Int.compare(i, j) of
    LESS => Br(t1, i, delete(t2, j))
    | GREATER => Br(delete(t1, j), i, t2)
    | EQUAL =>
      (case (t1, t2) of
          (Lf, _) => t2
        | (_, Lf) => t1
        | _ => let val (m, t2’) = delMin t2
          in Br(t1, m, t2’) end)
```

Expression Trees

```haskell
infix 6 ++ --;
infix 7 ** //;
```

```haskell
datatype fexpr =
  Const of real
  | X
  | ++ of fexpr * fexpr
  | -- of fexpr * fexpr
  | ** of fexpr * fexpr
  | // of fexpr * fexpr
```

Infix operators

```haskell
> datatype fexpr =
  con ++ : fexpr * fexpr -> fexpr
  con -- : fexpr * fexpr -> fexpr
  con ** : fexpr * fexpr -> fexpr
  con // : fexpr * fexpr -> fexpr
  con X : fexpr
  con Const : real -> fexpr
```

```haskell
> val it = [1, 3, 4, 5] : int list
```
Symbolic Differentiation  

\[ \text{fun } D(\text{Const } x) = \text{Const } 0.0 \]
\[ D \text{ } x = \text{Const } 1.0 \]
\[ D(f1 ++ f2) = (D f1) ++ (D f2) \]
\[ D(f1 -- f2) = (D f1) -- (D f2) \]
\[ D(f1 ** f2) = (D f1) ** f2 + f1 ** (D f2) \]
\[ D(f1 // f2) = ((D f1)**f2 -- f1 ** (D f2)) // (f2**f2) \]

Example:
\[ D (X ** (X **(Const 3.0 ++ X))); \]
\>
\[
\text{val it =} \\
++(**(Const 1.0, **(X, ++(Const 3.0, X))), \\
**(X, \\
++(**(Const 1.0, ++(Const 3.0, X)), \\
**(X, ++(Const 0.0, Const 1.0)))))) : \text{fexpr} \]

Expression: Computation of values

\[ \text{fun } \text{comp}(\text{Const } r,_) = r \]
\[ \text{comp}(X,y) = y \]
\[ \text{comp}(f1 ++ f2,y) = \text{comp}(f1,y) + \text{comp}(f2,y) \]
\[ \text{comp}(f1 -- f2,y) = \text{comp}(f1,y) - \text{comp}(f2,y) \]
\[ \text{comp}(f1 ** f2,y) = \text{comp}(f1,y) * \text{comp}(f2,y) \]
\[ \text{comp}(f1 // f2,y) = \text{comp}(f1,y) / \text{comp}(f2,y) \]

Example:
\[ \text{comp}(X ** (Const 2.0 ++ X), 4.0); \]
\>
\[
\text{val it = 24.0} : \text{real} \]

Abstract types

- internal representation is hidden from a user

Invariants may be protected by hiding the representation

Example: Violation of an invariant for search trees:
\[ \text{insert}(4, \text{Br}(\text{Lf}, 5, \text{Br}(\text{Lf}, 2, \text{Lf}))); \]
\>
\[\text{val it = Br(Br(Lf, 4, Lf), 5, Br(Lf, 2, Lf)) : tree}\]

Solution: Abstract data types
- search trees are only accessible by the following operations:

\[ \text{empty : stree} \]
\[ \text{insert} : \text{int} * \text{stree} -> \text{stree} \]
\[ \text{member} : \text{int} * \text{stree} -> \text{bool} \]
\[ \text{toList} : \text{stree} -> \text{int} \text{list} \]

Abstract types in SML: Search trees (1)

abstype stree = Lf | Br of stree * int * stree

with
\[ \text{val empty = Lf} \]

\[ \text{fun insert}(i, \text{Lf}) = \text{Br}(\text{Lf},i,Lf) \]
\[ \text{fun insert}(i, \text{tr as Br(t1,j,t2))) = \text{case Int.compare}(i,j) \text{ of} \]
\[ \text{.....} \]
\[ \text{fun member}(i, \text{Lf}) = \text{false} \]
\[ \text{fun member}(i, \text{Br}(t1,j,t2)) = \text{.....} \]

\[ \text{fun toList Lf = []} \]
\[ \text{fun toList(Br(t1,j,t2)) = toList t1 @ [j] @ toList t2; end;} \]
Abstract types in SML: Search trees (2)

The representation of stree is hidden:

```sml
> abstype stree
> val empty = <stree> : stree
> val insert = fn : int * stree -> stree
> val member = fn : int * stree -> bool
> val toList = fn : stree -> int list
```

Constructors are invisible:

```sml
Br(Lf, 5, Br(Lf,0,Lf));
```

— the search tree invariant cannot be violated

Abstract types in SML: Search trees (3)

Examples:

```sml
val st1 = insert(2, empty);
> val st1 = <stree> : stree

val st2 = insert(3, insert(7, st1));
> val st2 = <stree> : stree

member(4, st2);
> val it = false : bool

member(7, st2);
> val it = true : bool

toList st2;
> val it = [-3, 2, 7] : int list
```

Summary of chapters 7 and 8

- **datatype declarations** that give rise to types of **tagged values**
  - recursive
  - parameterized (giving polymorphic constructors)
  - they can be used to make:
    - disjoint union types
    - enumeration types, like `order` and `bool` (both predefined)
    - option types
    - tree types
- **abstype declarations** that give rise to **abstract data types**
- **case expressions**
- **patterns** for tagged values
- **exceptions**: how to declare, raise and handle
- **partial functions**: three ways of handling them in SML