Lecture 04: Programming Languages [CMPU-235]
September 12, 2017

Sameer S Pradhan
Vassar College
Standard ML
We can have some local declarations within an expression.

```ml
fun countSubordinates (Grunt _) = 0
  | countSubordinates (Manager m) =
    let
      val (_,grunts) = m
    in
      length grunts
    end
```

- Each declaration in scope from its point of definition until the `end`.
- Useful for naming intermediate results, and helper functions
Sometimes, useful to have mutually recursive datatypes:

datatype 'a evenlist = Empty | EvenCons of 'a * 'a oddlist
and 'a oddlist = One of 'a | OddCons of 'a * 'a evenlist

Similarly, can have mutually recursive functions:

fun evenlength Empty = 0
  | evenlength (EvenCons (_, ol)) = 1 + oddlength ol

and oddlength (One _) = 1
  | oddlength (OddCons (_, el)) = 1 + evenlength el
Anonymous and First Class Functions

In SML, as in other functional languages, functions may return functions, or take functions as arguments:

- fun addN n = fn x => n + x
- Take an \texttt{int}, return anonymous function that adds \textit{n} to \textit{x}
- Has type \texttt{int \to int \to int}
- fun modifyAge f (name, age) = (name, f age)
- Two patterns: match a thing and call it \(f\), match a person
- Can be given the type: 
  \[(\texttt{int \to int}) \to \texttt{person \to person}\]
Anonymous and First Class Functions

Example: \texttt{modifyAge (addN 1)} has type \texttt{person -> person}

\begin{itemize}
  \item \texttt{fun map f nil = nil}
  \item \texttt{| map f (x::xs) = (f x) :: (map f xs)}
\end{itemize}

This function has type

\begin{itemize}
  \item ('a -> 'b) -> 'a list -> 'b list
  \item \texttt{map (modifyAge (addN 1)) somePeople}
\end{itemize}
Exceptions

- Functions must deal with unexpected input. Sometimes there is no sensible result type.
- Sometimes one can modify the function to return an option type, and return `NONE` on bad input.
- However sometimes need truly exceptional behavior: no sensible way to deal with bad data locally.
Exceptions are declared by an exception declaration:

```ml
exception NegativeAge of person
```
Exceptions: Raising

Handling exceptions is done as follows:

```ml
canRentCar aPerson
  handle (NegativeAge p) => false
  | Div => false (* raised by integer divide by zero *)
```

Handle has some patterns that matches some exceptions. No need to handle all exceptions: unhandled ones propagate up to top level. May reraise an exception:

```ml
foo handle e => raise e
```
fun canRentCar (p as (_, age)) =  
  if age <= 0 then raise (NegativeAge p)  
  else age >= 25

Note I snuck in another pattern in here: \( y \text{ as } pat \) matches if \( pat \) matches the entire value, and also binds \( y \) to that value.
Full SML: a program is a collection of structures (*i.e.*, *modules*) and an expression to be evaluated

- Collaboration of multiple programmers
- Factoring of independent components
- Code reuse
A structure declaration is a namespace for type, variable and function definitions.

```ml
structure BinaryTree = struct
  datatype 'a tree = ...
  val empty = ...
  fun addElem (t, x) = ...
  fun map f t = ...
  fun isEmpty t = ...
  fun toList t = ...
end
```

Outside the structure, refer to `BinaryTree.empty`, etc.
Signatures as type of Structures

- Just as values have types, structures have a signature.
- Compiler will infer a *principal* signature for a structure that includes:
  - the definition of every type abbreviation and datatype
  - the declaration of every exception
  - the type of every value and function

```ml
sig
datatype 'a tree = ...
val empty : 'a tree
val addElem : 'a * 'a tree -> 'a tree
val mapTree : ('a -> 'b) -> 'a tree -> 'b tree
...
end
```
Abstraction: non-principal Signatures

You can also write down less specific signatures:

- Hide the definition of types or datatypes
- Abstract data types
- Hide helper functions

```ml
signature BINARY_TREE = sig
  type 'a tree
  val empty : 'a tree
  val addElem : 'a * 'a tree -> 'a tree
  val map : ('a -> 'b) -> 'a tree -> 'b tree
end
```
Abstraction: Hiding the Implementation

```plaintext
structure BinaryTree :> BINARY_TREE = struct
    datatype 'a tree = ...
    val empty = ...
    fun addElem (t, x) = ...
    fun map f t = ...
    fun isEmpty t = ...
    fun toList t = ...
end

Outside the implementation, the representation of trees and several functions are inaccessible.
```
To facilitate code reuse, possible to parametrize the implementation of a structure by zero or more other structures:

functor BalancedBinaryTreeFn (structure B : BINARY_TREE) =
  struct
    fun balance t = ... (* mentions B.addElem, etc *)
  end

Then use as:

structure MyBBT = BalancedBinaryTreeFn (structure B = BinaryTree)
structure BetterBBT = BalancedBinaryTreeFn (structure B = BetterBinaryTree
What I have not told you...

- Anything about side-effects: I/O, mutable store, concurrency, etc.
  - Turns out you can accomplish a lot (nearly everything for this course) without them.
- The truth about equality types.
  - But you can mostly pretend they don’t exist.
- Record types: like tuples with named components
- Projection functions for tuples and records
What I have not told you...

- Advanced modular programming: substructures, sharing specs, where-type
- Useful library functions.
  - Read the SML Basis documentation on the SML/NJ webpage www.smlnj.org
- Compilation management
  - aka CM, documented on the SML/NJ webpage. Will provide instructions in the programming assignments.
Further Reading

- Ullman, “Elements of ML Programming”
- Paulson, “ML for the Working Programmer”
- Harper, “Programming in Standard ML”
- SML/NJ webpage
- Our course website