Functional Programming in ML Overview

Next couple of weeks. Sethi chapters 8, 9.

- Primitive types, records, lists, tuples
- Programming in ML:
  - Recursive functions
  - Pattern matching
- Type definition
  - Recursive datatypes
- String manipulation
- Type checking:
  - Type variables
  - Parametric polymorphism
  - Overloading
- Exception handling
- Summary.

ML and Typechecking

ML - Meta Language. Began as the programming language for a machine-assisted system for formal proofs.
Based explicitly on Lambda Calculus.

- Functions are first-class objects. But better than Scheme since has a more reasonable syntax (less parentheses) and pattern-matching (but less than in Prolog)
- Strongly typed (Scheme is untyped). Static typechecking is done at compile time. Has abstract data types

Reading - Sethi, Ch. 8-9.
Playground - Standard ML of New Jersey.

```
eddie% sml
Standard ML of New Jersey, Version 0.93, ...
val it = () : unit
-
```


Types: Values and Operations

A *type* consists of a set of elements called *values* together with a set of functions called *operations*.

Primitive types - values are atomic, i.e., there is no external structure:
- boolean (bool) - true, false
- string - "red", "white", "blue"

(note that strings are pretty complex data types in C or C++)
- integer - 1, 10, ...
- real - 0.0, 3.14, ...

All of the above are built-in types.

The operations defined on booleans and strings are equality (=) and inequality (<>). Operations defined on integers and reals are the usual arithmetic and logic operations.

Product of Types (Tuple)

Used to represent non-homogeneous collection of data, like structs in C.

Syntax: (<elem1>, <elem2>, ...).

Eg., `(1, "one")`

Projection functions - extract elements from pair (more on those later). These functions are *not* predefined for tuples, but we can define them easily:

```
fun first(x,y) = x;
fun second(x,y) = y;
```

Type of tuple - product of types of its elements. Type of `(1, "white")` is `int * string`.

Examples:

```
first (1, "one")
  val it = 1 :  int
second (1, "one")
  val it  "one" :  string
```

Records

A record is a tuple with named fields. For example, `(real, real)` can be represented as `{re:real, im:real}`. We can refer to fields of the record using the field name:

`# <field label> <record value>`

Examples:

```
{name="Jane Doe", age = 28, married =true };
  val it = {age=28,married=true,name="Jane Doe"}
    age:int, married:bool, name:string}
```

ML rearranges record using lexical order of fields.

`it` keeps result of the last computation and can be accessed directly.

```
#name it
  val it = "Jane Doe" : string
```

More familiar syntax, i.e., `it.name` is reserved for other purposes (field in a module).

Variables

Have usual names - `a`, `b`, `c`.

Variables do not need to be declared before they are used.

Variables denote values (like a name for a location in which some data is stored)

Lists

A *list* is a finite-length sequence of homogeneous elements. The type `A list` consists of all lists of elements, where each element belongs to type `A`.

E.g. `int list` - consists of all lists of integers

List elements are written between brackets `[` and `]` and are separated by commas. The list with no elements (empty list) is written equivalently as `[ ]` or as `nil`.

Examples of lists:

```
[1, 2, 3] ⇒ int list
["red", "white", "blue"] ⇒ string list
[ [ 1, 2 ], 3] ⇒ ERROR
[ 1, [ 2, 3 ] ] ⇒ ERROR
[ [ 1, 2 ], [ 3, 4 ] ] ⇒ int list list
[ [ 1, [ 2, 3 ] ], 4, [ 5, 6 ] ] ⇒ ERROR
```


Operations on Lists

Operations on lists must be prepared to construct and inspect lists of any length.

Some operations:

- `null(x)`

True if `x` is the empty list and false otherwise

- `hd(x)`

The first or head element of list `x`

- `tl(x)`

The tail or rest of the list after the first element is removed.

- `a :: x`

Construct a list with head `a` and tail `x`

- `a @ b`

Append (concatenate) list `b` at the end of list `a`.

List Operations - examples

Every list can be constructed using :::

```
[ 1, 2, 3 ] = 1 :: [ 2, 3 ] =
= 1 :: 2 :: [ 3 ] = 1 :: 2 :: 3 :: []
```

Examples:

```
1 :: [2, 3] => [1, 2, 3]
hd [1, 2, 3] => 1
tl [1, 2, 3] => [2, 3]
tl [[1, 2, 3]] => []
[1] :: [2, 3] => ERROR
[1] @ [1] => [1, 1]
[1] @ [[2]] => ERROR
hd [[1], 2, 3] => ERROR
hd [[1, 2, 3]] = > [1, 2, 3]
```


Functions

Functions are from Domain to Range.

Type `A` $\rightarrow$ `B` - set of all functions from `A` to `B`.

A function is *total* if it is defined for each element of `A`.

Otherwise, the function is *partial*

To apply a function - `f a`. Parentheses do not matter, so

```
f a = (f a) = f(a)
```

Application is left-associative:

```
f a b = (f a) b
```

`f a = b` means that `b` is the value of `f` at `a`

More on Types

structured types:

type	constructor	notation	example
function	$\rightarrow$	infix	`int -> bool`
product	$*$	infix	`int * int`
list	`list`	postfix	`string list`

$->$ has higher precedence than $*$ which has higher precedence than `list`.

To define a new basic type - enumerate its values (`enum` in C):

```
datatype direction = ne | se | sw | nw;
```

Here we listed constructors for this type. | stands for "OR", separating enumerated items.

To give a name to type - keyword `type`

```
type intpair = int * int
type di = direction * int
```


ood for slide eater