Lambda Terms

ML needs a notation to write down unnamed (anonymous) functions, similar to the lambda expressions Scheme uses. That notation is

\( \text{fn arg => body}; \)

For example,

\[
\text{val sqr = fn x : int => x*x;}
\]

\[
\text{val sqr = fn : int -> int}
\]

In fact the notation used to define functions,

\[
\text{fun name arg = body;}
\]

is actually just an abbreviation for the more verbose

\[
\text{val name = fn arg => body;}
\]

An anonymous function can be used wherever a function value is needed. For example,

\[
\text{map (fn x => [x]) [1,2,3];}
\]

\[
\text{val it = [[1],[2],[3]] : int list list}
\]

We can use patterns too:

\[
\text{(fn [] => [])}
\]

\[
\text{|(h::t) => h::h::t;}
\]

\[
\text{val it = fn : 'a list -> 'a list}
\]

(What does this function do?)

Polymorphism vs. Overloading

ML supports polymorphism.

A function may accept a polytype (a set of types) rather than a single fixed type.

In all cases, the same function definition is used. Details of the supplied type are irrelevant and may be ignored.

For example,

\[
\text{fun id x = x;}
\]

\[
\text{val id = fn : 'a -> 'a}
\]

\[
\text{fun toList x = [x];}
\]

\[
\text{val toList = fn : 'a -> 'a list}
\]

Overloading, as in C++ and Java, allows alternative definitions of the same method or operator, with selection based on type.

Thus in Java \( + \) may represent integer addition, floating point addition or string concatenation, even though these are really rather different operations.

In ML \( +, -, *, \) and \( = \) are overloaded.

When \( = \) is used (to test equality), ML deduces that an equality type is required. (Most, but not all, types can be compared for equality).

When ML decides an equality type is needed, it uses a type variable that begins with two tics rather than one.

\[
\text{fun eq(x,y) = (x=y);}
\]

\[
\text{val eq = fn : ''a * ''a -> bool}
\]
Defining New Types in ML

We can create new names for existing types (type abbreviations) using

```ml
val type id = def;
```

For example,

```ml
val type triple = int*real*string;
val type rec1 = {a:int,b:real,c:string};
val type rec2 = {a:int, b:real, c:string}
val type 'a triple3 = 'a*'a*'a;
val type intTriple = int triple3;
```

These type definitions are essentially macro-like name substitutions.

The Datatype Mechanism

New types are defined using the `datatype` mechanism, which specifies new data value constructors.

For example,

```ml
val datatype color = red|blue|green;
```

Pattern matching works on user-defined types using their constructors:

```ml
fun translate red = "rot"
| translate blue = "blau"
| translate green = "gruen";
```

```ml
val translate = fn : color -> string
```

```ml
fun jumble red = blue
| jumble blue = green
| jumble green = red;
```

```ml
val jumble = fn : color -> color
translate (jumble green);
```

```ml
val it = "rot" : string
```

SML Examples

Source code for most of the SML examples presented here may be found in

```ml
~cs538-1/public/sml/class.sml
```

Parameterized Constructors

The constructors used to define data types may be parameterized:

```ml
val datatype money = none
| coin of int
| bill of int
| iou of real * string;
```

```ml
val datatype money =
  bill of int | coin of int
| iou of real * string | none
```

Now expressions like `coin(25)` or `bill(5)` or `iou(10.25,"Lisa")` represent valid values of type `money`. 
We can also define values and functions of type \texttt{money}:

\begin{verbatim}
val dime = coin(10); val dime = coin 10 : money
val deadbeat = iou(25.00, "Homer Simpson"); val deadbeat = iou (25.0, "Homer Simpson") : money

fun amount (none) = 0.0
| amount (coin (cents)) = real (cents) / 100.0
| amount (bill (dollars)) = real (dollars)
| amount (iou (amt, _)) = 0.5 * amt;

val amount = fn : money -> real
\end{verbatim}

Polymorphic Datatypes

A user-defined data type may be polymorphic. An excellent example is

\begin{verbatim}
datatype 'a option = none | some of 'a;
datatype 'a option = none | some of 'a
val zilch = none;
val mucho = some (10e10);
val mucho = some 100000000000.0 : real option

type studentInfo = {name:string, ssNumber:int option};
type studentInfo = {name:string, ssNumber:int option}

val newStudent = {name="Mystery Man", ssNumber=none} : studentInfo;
\end{verbatim}

Datatypes may be Recursive

Recursive datatypes allow linked structures without explicit pointers.

\begin{verbatim}
datatype binTree = null | leaf | node of binTree * binTree;
datatype binTree = leaf | node of binTree * binTree | null

fun size (null) = 0
| size (leaf) = 1
| size (node (t1, t2)) = size (t1) + size (t2) + 1
val size = fn : binTree -> int
\end{verbatim}
Recursive Datatypes may be Polymorphic

datatype 'a binTree =
  null | leaf of 'a | node of 'a binTree * 'a binTree

datatype 'a binTree =
  leaf of 'a | node of 'a binTree * 'a binTree | null

fun frontier(null) = []
  | frontier(leaf(v)) = [v]
  | frontier(node(t1,t2)) = frontier(t1) @ frontier(t2)

val frontier = fn : 'a binTree -> 'a list

We can model n-ary trees by using lists of subtrees:

datatype 'a Tree =
  null | leaf of 'a | node of 'a Tree list;

datatype 'a Tree = leaf of 'a | node of 'a Tree list | null

fun frontier(null) = []
  | frontier(leaf(v)) = [v]
  | frontier(node(h::t)) = frontier(h) @ frontier(node(t))
  | frontier(node([])) = []

val frontier = fn : 'a Tree -> 'a list