CS109A ML Notes for the Week of 1/16/96

Using ML

ML can be used as an interactive language. We shall use a version running under UNIX, called SML/NJ or "Standard ML of New Jersey."

- You can get SML/NJ by the command `sml` on the "elaine's."
- It is also possible to run a program without interaction. Put the program in a file, e.g. `foo` and issue the UNIX command

```
sml <foo
```

Example: Here is an example of an interaction. Human-typed things are in the `teletype` font; things typed by the machine are in *italic* font.

`sml`
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit

`5;`
val it = 5 : int

`"abc";`
val it = "abc" : string

`<ctrl>d`
(machine returns to UNIX command level)

- The normal response of SML/NJ is `val` (short for "value"), followed by `it` (a special identifier that means "the previously typed expression"), an equal-sign, the value of the expression, a colon, and the type of the expression.
- Special case: the first response says that `it` has the value `()`, and its type is `unit`. The unit is a special "null type," whose only possible value is `()`.
- We then type the expression 5, followed by a semicolon. ML responds that the value of this expression is 5 and it is an integer.
  - Semicolon must end all expressions.

- ☐ SML/NJ gives you a "–" prompt when it is ready to begin an expression and an "=" prompt if it is waiting for you to complete an expression. Often an unexpected "=" means you have forgotten the semicolon.

- We type `"abc"`, and ML tells us this expression is a string with value `"abc"`.

Variables in ML

An ML program operates in a workspace of variables, much like a C program. We can assign a value to variables `foo` and `bar` by

```
val foo = 5;
```
val foo = 5 : int

```
val bar = 7;
```
val bar = 7 : int

- Remember to use "val" as if saying "the value of foo is 5."
- ML tells the value of the variable, not "it."
- We can use variables in expressions, as in other languages. ML evaluates any expression it is given.

```
foo + bar;
```
val it = 12 : int

Arithmetic Operators

Usual +, –, *, /.

- But / is for reals; use `div` for integers.
- `mod` gives the remainder of integers.
- `~` denotes unary minus.

Example:

```
4.0+5.0;
```
val it = 9.0 : real

```
30 div 7;
```
val it = 4 : int

```
30 mod 7;
```
val it = 2 : int

```
~3*(~4);
```
val it = 12 : int

- Note that parens (or a space) are needed for the last example. ML would interpret `~3*~4` as if `*~` were a single operator and complain that it had never heard of that operator.

Concatenation of Strings

Operator `^` denotes concatenation of strings.

```
"foo" ^ "bar"
```
val it = "foobar" : string

Comparison Operators

As in C, but `!=` ⇒ `<>` and `==` ⇒ `=`.

```
4<=3;
```
val it = false : bool

```
"love" < "war";
```
val it = true : bool

- Note comparison of strings is lexicographic (dictionary) order.
- Type `bool` (Boolean) is the type of the result of a comparison. This type has only the two values: `true` and `false`.

Logical Operators

`&&` ⇒ `andalso`; `||` ⇒ `orelse`; `!` ⇒ `not`.

```
3<4 andalso 5<4;
```
val it = false : bool

```
3<4 andalso (not (4<5) orelse 5<6);
```
val it = true : bool

- Precedences of logical operators relative to each other and to the arithmetic or comparison operators are as in C, with one exception (not made clear in the book):
  - □ `not` has higher precedence than any infix operator. Thus, the parens in "`not (4<5)`" are essential. Without them, ML tries to apply `not` to 4, and complains that it cannot apply a this Boolean operator to an integer.

If-Then-Else Operator

if-then-else is used like `?:` in C.

- It is an expression-operator, not a statement as is "if-else" is in C.

```
if 3<4 then 5 else 6;
```

val it = 5 : int

Types

Four basic types: `int`, `real`, `bool`, `string`.

- Values are denoted as in C, but
  - □ `bool` has only values `true` and `false`.
  - □ `real` in ML is `float` in C.
  - □ `string` is a basic type in ML, not an array of characters as in C.

Types Must Agree

ML will figure out the type for most expressions, using clues such as the types of arguments.

- But there is no automatic coersion, as from int to float in C.

Example:

```
3 + 4.0;
```

std_in:2.1-2.7 Error: operator and operand don't agree (tycon mismatch)
*operator domain: int * int*
*operand: int * real*
in expression:
+ : overloaded (3,4.0)

- ML views every operator as applying to a single operand. Even a binary, infix operator is thought of as applying to a pair, e.g. the pair $(3, 4.0)$ of type `int * real`.

- Many ML operators like `*` are *overloaded*; they can apply to operands of various types, in the case of `*` to either a pair of integers, a pair of reals, or a pair of types.

  - ☐ Notice that `*` in addition to its arithmetic role also is used to build structure-types, such as pair-types in this example.

- When ML sees the `3` and then the `*`, it assumes that the `int * int` version of `*` is meant. ML complains when its operand turns out to be of type `int * real`.

- I think that the line numbers in SML/NJ error messages are too high by 1. "std_in:2.1-2.7" is supposed to mean that the error occurs in characters 1–7 of line 2, but in this example, there was only one line of input.

Coercion

There are a number of operators that convert from one type to an "equivalent" value in another type.

- See pp. 17–18 and 249–250 of EMLP.

Example:

```
3.14159 * real(2);
```
val it = 6.28318 : real

```
floor(3.14159);
```
val it = 3 : int

```
ord("#");
```
val it = 35 : int

```
chr(35);
```
val it = "#" : string

ML Identifiers

Names of variables in ML may be formed in one of two ways:

1. *Alphanumeric* identifiers, like identifiers in C, but the apostrophe `'` may also be used as a letter.

   - □ However, an identifier *beginning* with `'` may only have a type as a value, not an "ordinary" value."

2. *Symbolic* identifiers are strings composed of 20 different symbols, mostly the usual operator symbols (see p. 20 of EMLP for complete list).

   - □ Thus, ordinary operator names like `*` or `<=` are symbolic identifiers. So would `*~`, which explains why `3*~4` is not interpreted "correctly."

Tuples

A *tuple* is a parenthesized list of values of any type.

- Tuples are like structs in C, but without component names (but ML also has the ability to name components as we shall see much later).

  ```
  (4, 4.0, "four");
  ```
  *val it = (4,4.0,"four") : int * real * string*

- Note that the type of a tuple is the list of the types of its components separated by `*`'s.

We extract a component of a tuple with the `#`i operator; i is any integer for which there is a component.

```
#2(4, 4.0, "four");
```
val it = 4.0 : real

Lists

A *list* is a sequence of values surrounded by square brackets and separated by commas.

- Unlike tuples, which use round rather than square brackets, the elements of a list *must* have the same type.

```
["a", "b", "c"];
```
val it = ["a","b","c"] : string list

```
[(1,2), (3,4)];
```
*val it = [(1,2),(3,4)] : (int * int) list*

- The type of the first list is `string list`, i.e., a list of strings. The second list is a list of pairs of integers.
- Note: The empty list is denoted by `[]` or `nil`.

Operators on Lists

`hd` and `tl` extract the *head* (first element) and *tail* (list of the remaining elements).

```
hd([1,2]);
```
val it = 1 : int

```
tl([1,2]);
```
val it = [2] : int list

```
tl [1];
```
val it = [] : int list

- Note the type of the head is the type of an element, while the type of the tail is a list of elements.
- Notice in the last example that parentheses are not needed for arguments of one-argument functions in ML.

`::` is the *cons* operator; it connects a head and a tail to form a new list.

```
1::[2,3];
```
val it = [1,2,3] : int list

`@` is the *concatenation* operator for lists (not for strings, where `^` is used).

```
[1]@[2,3];
```
val it = [1,2,3] : int list