Chapter 10c – ML Supplemental
Part 1: Introduction
Imperative Languages

- Design of imperative languages is based directly on the von Neumann architecture

von Neumann Architecture
(Adapted from Aspray, 1990a)
Imperative Languages (cont.)

- Programs in imperative languages rely heavily on modifying the values of a collection of variables, called the state.
- Before execution, the state has some initial value $\sigma$.
- During execution, each command changes the state.

\[ \sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n = \sigma' \]
Example

• In a sorting program, the state initially includes an array of values
• When the program has finished, the state has been modified in such a way that these values are sorted
• Intermediate states represent progress towards this goal
Assignment

• The state is typically modified by assignment commands
• By using control structures, one can execute these commands conditionally, or repeatedly, depending on other properties of the current state
Factorial: imperative language

n := x;
a := 1;
while n>0 do
begin
    a := a * n;
    n := n – 1;
end;
...

Declarative Languages

- The declarative languages have no state
- The emphasis is placed entirely on programming with expressions
- *Functional Languages*: The underlying model of computation is function
- *Logic Programming Languages*: The underlying model of computation is predicates
Declarative Languages

- *State-oriented* computations are accomplished by carrying the state around explicitly rather than implicitly
- *Looping* is accomplished via recursion rather than sequencing
Factorial: declarative language

\[
\text{fac } x \ 1 \\
\text{where fac } n \ a \\
\quad = \text{ if } n > 0 \ \text{then } \text{fac}(n-1) \ (a*n) \\
\quad \quad \text{else } a \\
\]

- \( n \) and \( a \) are examples of carrying the state around explicitly
- recursive structure mimics the looping behavior
What vs. How

- The value of the program is the desired factorial
  - rather than storing it in a store
- Declarative programming is often described as expressing what is being computed rather than how
Functional Programming

• Functions are first class values
  ▫ a data type is first class if it can be used anywhere, passed to and returned from functions, assigned to variables

• May also have *imperative* constructs
  ▫ Examples: Lisp, Scheme, ML, Erlang

• Pure functional languages have no implicit side-effects or other imperative features
  ▫ Examples: Miranda, Haskell
Backus’ Turing Award

- John Backus was designer of Fortran, BNF, etc.
- Turing Award Lecture in 1977
  - Functional prog better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws
    Reason about programs
    Optimizing compilers
Von-Neumann Bottleneck

- Backus' famous paper encouraged much interest in functional languages
  - Breaking the Von-Neumann bottleneck
- Von Neumann bottleneck: pumping single words back and forth between CPU and store
- Task of a program: change store in some major way
Von-Neumann Bottleneck

• It has kept us tied to word-at-a-time thinking
  ▫ Instead of encouraging us to think in terms of the larger conceptual units of the task at hand

• The assignment statement is the von Neumann bottleneck of programming languages

• Pure functional programming languages remove state and assignments

• Concurrency possible: order of evaluation doesn’t matter
Referential Transparency

- A referentially transparent function is a function that, given the same parameter(s), always returns the same result.
- Do we have such property in imperative languages?
Referential transparency

- If the function has side-effects (updating a global variable, doing input or output), then $f(3) + f(3)$ may not be the same as $2 * f(3)$.
  - The second $f(3)$ has a different meaning than the first
- Purely declarative languages guarantee referential transparency
- The importance of referential transparency is that it allows a programmer (or compiler) to reason about program behavior
• “Start with programming languages as an abstract notation for specifying algorithms and then work down to the hardware.” (Dijkstra 1976)
Part 2: SML/NJ
SML/NJ

- Standard ML of New Jersey system is used in many courses in several universities:
  - CPSC 201: Introduction to Computer Science – Yale
  - CS312 - Data Structures and Functional Programming – Cornell
  - CS15-212: Principles of Programming – CMU
  - CS15-312: Foundations of Programming Languages – CMU
  - CS510: Programming Languages – Princeton
  - CS421/521: Compilers and Interpreters – Yale
  - CS345: Programming Languages – UT Austin
  - CS 442: Principles of Programming Languages – Waterloo
  - CS153: Principles of Programming Language Compilation – Harvard
ML : History

- ML comes from "Meta Language"
- It was originally designed to be the programming language for Robin Milner's LCF theorem proving system
- LCF stands for Logic of Computable Functions, since it implemented a logic of that name due to Dana Scott
ML : History (cont.)

- Today, LCF is of only historical interest, while ML has become quite important
- ML has been used in many different kinds of applications
  - Because it has a good compiler, strong typing, and a good module system
- Some attempts to add ML to .NET
Introduction to ML

- Highlights
  - **Functional Language**
    - Functions are pervasive:
    - First-class values
      - Storable, arguments, results, nested
  - **Strongly-typed language**
    - Every expression has a type
    - Certain errors cannot occur
    - Polymorphic types provide flexibility
  - **Flexible Module System**
    - Abstract Types
    - Higher-order modules (functors)
Introduction to ML (cont.)

• Statically Scoped Language
  ▫ Compile-time resolution of values
• Call-by-value language
  ▫ f(e)
• High-level programming features
  ▫ Data types
  ▫ Pattern matching
  ▫ Exceptions
  ▫ Mutable data discouraged
Interacting with ML

- Interactive Language
  - Type in expressions
  - Evaluate and print type and result
- The top level ML prompt is “-”
- As ML reads a phrase it prompts with “=” until a valid ML statement is found
Example

Since no identifier is given to bind to the value, the interactive system has chosen the identifier *it* and bound it to the result of 4+6.

The semicolon (";";) is a marker that indicates to the SML/NJ system that it should perform the interactive top-level loop.
The interactive top-level loop

- **elaborate**
  - that is, perform typechecking and other static analyses
- **compile**
  - to obtain executable machine code
- **execute**
- **print** the result of this declaration
Declarations

- The declaration `val x=e` evaluates `e` and binds the resulting value to `x`

  ```ml
  val x = 2 * 3;
  val x = 6 : int
  ```

- A declaration `d` can be made local to evaluation of an expression `e` by evaluating the expression `let d in e end`
Functions

- A function $f$ with formal parameter $x$ and body $e$
  - `fun f x = e`
- Apply the function $f$ to an actual parameter $e$
  - $f e$

```
- fun f x = 3*x;
val f = fn : int -> int
- f 2;
val it = 6 : int
```
Lambda Expressions

- The expression `fn x=>e` is equivalent to `λx.e`

```
- val f = fn x=> 5*x;
val f = fn : int -> int
- f 4;
val it = 20 : int
- (fn y => (fn x=>x) y) 3;
val it = 3 : int
```
Pattern matching

- Functions can be defined by pattern matching
- Suppose function $f$ is defined by:
  \[
  \text{fun } f \ p_1 = e_1 \\
  \mid \text{fun } f \ p_2 = e_2 \\
  \vdots \\
  \mid \text{fun } f \ p_n = e_n
  \]
- An expression $f \ e$ is evaluated by successively matching the value of $e$ with the pattern $p_1, p_2, \ldots p_n$
Example

- Consider the Fibonacci function
  ```ml
  fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2);
  ```
- Evaluation of `fib 5` causes 5 to be matched with 0 and 1, both of which fail, and the with `n` which succeeds
Types

- Standard ML is strongly typed
- Programs are type checked before they are run
- The type checking system is static
  - The types are checked when the program is compiled
Unit

- There is one special type called **unit**, written ()
- We use the unit type when we are interested in some expression for its side-effects, rather than for its value
- A common example of this is input/output

```
- val str = "salam";
  val str = "salam" : string
- print str;
  salam
  val it = () : unit
```
Bool

- There are two values of type bool, `true` and `false`
- The most common expression associated with Boolean is conditional
  - `if e1 then e2 else e3`
- There is no if-then without else
  - A conditional expression must have a value whether the test is true or false
Other Common Types

- $0, 1, 2, \ldots, -1, -2, \ldots$ : int
- $1.0, 2.0, 3.3333, \ldots$ : real
  - $4+4.0$ is wrong in ML
- “Marjan Sirjani” : string
  - String concatenation : $^\wedge$
  - “Marjan” $^\wedge$ “Sirjani” $\equiv$ “MarjanSirjani”
Tuples

- Tuples may be formed of any types of values
- Tuples are written with parentheses
- Tuples types are written with *

- (3,6,false,"Rebeca");
  val it = (3,6,false,"Rebeca") : int * int * bool * string

- #2("Formal","Method","Lab");
  val it = "Method" : string
Records

- Records are similar to tuples but have names for each of the elements instead of numbered positions
- Records values and record types are written with curly braces

```ml
- {First_name="Hossein",Second_name="Hojjat",id_numer=81018448};
val it =
{First_name="Hossein",Second_name="Hojjat",id_numer=81018448} :
{First_name:string, Second_name:string, id_numer:int}
```
Records (cont.)

- Record components can be accessed by # function and the component name

```ml
#Last({First="Edsger",Middle="Wybe",Last="Dijkstra"});
val it = "Dijkstra" : string
```
Lists

• A list consists of a series of items of the same type

- [2,4,5];
val it = [2,4,5] : int list
- [2,"4",5];  
stdIn:28.1-28.10 Error: operator and operand don't agree [literal]
  operator domain: string * string list
  operand: string * int list
  in expression:
    "4" :: 5 :: nil
- [fn x => x+2, fn y => 2];
val it = [fn,fn] : (int -> int) list
Lists Operators

• The cons operator :: takes an element and attach it to a list of that same type
• It actually constructs a list
  ▫ takes an item and a list and returns a list

- 4::nil; val it = [4] : int list
- 3::8::5::it; val it = [3,8,5,4] : int list
Lists Operators (cont.)

- **hd** returns the head of the list
  - that is the first element of a list
- **tl** returns the tail of a list
  - that is everything except the first element

- `hd["ali","naghi","mali"]`; val it = "ali" : string
- `1::tl[1,2,3]`; val it = [1,2,3] : int list
Lists Operators (cont.)

- The append operator @ is used to join two lists together

```
- [1,2] @ [3,4];  
val it = [1,2,3,4] : int list
- [1,2] @ ["formal"];  
stdIn:37.1-37.19 Error: operator and operand don't agree [literal]
  operator domain: int list * int list
  operand: int list * string list
  in expression:
    (1 :: 2 :: nil) @ "formal" :: nil
```
Polymorphism

- Consider the identity function : \( \text{fn } x \Rightarrow x \).
- The body of this function places no constraint on the type of \( x \).
  - The identity function is polymorphic.

\[
\begin{align*}
- \text{fn } x \Rightarrow x; \\
\text{val it = fn : 'a -> 'a}
\end{align*}
\]

- Here, ‘\( a \)’ is called a type variable, and ‘\( a \to a \)’ is called a type scheme.
Polymorphism (cont.)

- The type variables stand for an unknown, but arbitrary type expression
  - They are written ‘a (pronounced alpha), ‘b (pronounced beta), ‘c (pronounced gamma)
- A type scheme is a type expression involving one or more type variables
- For example: the instances of ‘a->‘a can be:
  - int -> int, string -> string, (int*int) -> (int*int)
- but not : int -> string
map : a useful function

- **map** type:
- **fn** : ('a -> 'b) -> 'a list -> 'b list

- a function \( f \) with argument type 'a and return type 'b
- a list \( l \) of elements of type 'a
map example

- It returns a list obtained by applying $f$ to each element of $l$
  - which is a list of elements of type 'b

- val double = fn x => 2 * x;
val double = fn : int -> int
- map double [1,2,3,4];
val it = [2,4,6,8] : int list
Type Constructors

- Both `list` and `*` are examples of type constructors
- `list` has one argument
  - `‘a list`
- `*` has two argument
  - `‘a * ‘b`
- Type constructors may have various predefined operations associated with them
  - For example `list` has null, hd, tl, …
Data Type Declaration

- A datatype declaration introduces
  1. One or more new type constructors. The type constructors introduced may, or may not, be mutually recursive
  2. One or more new value constructors for each of the type constructors introduced by the declaration
Data Type Declaration

datatype student = BS | MS | PHD;

- student is a type constructor
- student has three value constructors. Value constructors correspond to constructors of object-oriented languages
Data Type Declaration

- datatype student = BS | MS | PHD;
  datatype student = BS | MS | PHD
- val ehsan = BS;
  val ehsan = BS : student
- val hossein = MS;
  val hossein = MS : student
- val ramtin = PHD;
  val ramtin = PHD : student
Constructors Arguments

- The **type constructor** may take zero or more arguments
  - a zero-argument or nullary type constructor is just a type
- The **value constructor** may take zero or more arguments
  - a zero-argument or nullary value constructor is just a constant
datatype 'a option = NONE | SOME of 'a;

option is a unary type constructor

NONE is a nullary value constructor

SOME is a unary value constructor

For example, some values of string option are NONE, SOME “salam”
Data type example

- datatype number = exact of int | approx of real | NULL;
- datatype number = NULL | approx of real | exact of int
- val exactValue = exact 3;
- val exactValue = exact 3 : number
- val approxValue = approx 1.9;
- val approxValue = approx 1.9 : number
- val zero = NULL;
- val zero = NULL : number

Using the value constructor
Enumerated Type

- If none of the constructors have associated data, we get something analogous to an enumerated type in other languages

```
datatype direction = north | east | south | west;
```
Tagged Unions

- The type constructors in ML are sometimes called "tagged unions"
- A union data type (efficient, small) that, unlike with C/C++, come along with a tag field that tells what's in the union
Recursive Data T

- datatype 'a tree =
  Empty |
  Node of 'a tree * 'a * 'a tree;
1. The empty tree Empty is a binary tree
2. If tree_1 and tree_2 are binary trees, and val is a value of type typ, then Node( tree_1, val, tree_2) is a binary tree
Recursive Data type (example)

- Node (Empty,3,Empty);

- Node(Node(Empty,4,Empty),5,Node(Empty,3,Empty));
Defining Exceptions

- Exception declaration
  - Type of data that can be passed in exception
    - ML: exception <name> of <type>
    - C++/Java: any data type

- Raising an exception
  - Abort the rest of current block and jump out
    - ML: raise <name> <arguments>;
    - C++: throw <value>;

- Handling an exception
  - Continue normal execution after exception
    - ML: <exp1> handle <pattern>=><exp2>; ...
    - C++: try { …} catch (<type> var) {…} …
Structures

- A structure is a unit of program
- It consists of a sequence of declarations of types, exceptions and values
- The declarations are enclosed in keywords `struct` and `end`
- The result can be bound to an identifier using a `structure` declaration
Structure Example

structure Complex =
  struct
    type t  = real * real;
    val zero = ( 0.0, 0.0);
    fun sum (((x,y), (x',y')))= ( x + x', y + y') : t;
    fun diff (((x,y), (x',y')))= ( x - x', y - y') : t;
  end;

- Complex.sum( (1.0,2.0), (3.0,4.0));
val it = (4.0,6.0) : Complex.t
Opening a Structure

- An open declaration has the from
- open strid₁, strid₂,…,stridₙ
- It incorporates the body of the structures into the current environment
- We may use the declarations without referring to the structure
• There are lots of things left to learn from SML
• You have to learn the remaining yourselves!
• Don’t forget your programming project!
Some Good Books to read

- *Elements of ML Programming*, Jeffrey Ullman, Stanford University