Functional Programming with ML

- ML is a functional language like Scheme
  - Very different from Scheme in syntax and use
  - Can be pure – no assignment
  - ML stands for Meta Language
  - Originally developed for theorem proving (1978, revised 1997)
  - Very popular in Europe

- ML characteristics
  - Static type checking – types are very important
  - Strong type checking – program is "safe" if interpreter accepts
  - "variables" must be declared
  - Syntax more like C than Scheme

SML – Standard ML

- Popular version of ML
  - Available for various platforms
  - www.smlnj.org

- Interpreted (interactive like Scheme)
  - There are compilers for ML
  - Run from command line environment: sml
  - Prompt is "-"
  - Terminate statements with a semi-colon
  - Ctrl-Z (Windows) or Ctrl-D (Unix/Linux) to quit
ML Basics

- Case sensitive, infix notation
- Primitive data types built in
  - Numbers – integer or floating point (int or real)
  - Strings – double quotes like Java (string is a type in ML)
  - Character, e.g., "a" (char is the type name)
  - Booleans - true and false (bool is the type name)
- Identifiers can be letters, digits, underscores, can't begin with digit
- Symbols can also be identifiers, and many are built in (e.g., +)
- Comments in ML are delineated by (* and *) and may be nested

Invoking ML

- Command line interpreter, start by typing sml
- Assumes sml executable is in PATH
- Interpreter prints a prompt of "-"
- Secondary prompt of "=" is given if you have not typed a complete statement (this allows multiple line input)
- Interpreter evaluates each statement and prints the result
  - -13;
    val it = 13 : int
  - 3 * 13;
    val it = 39 : int
  - it + 7;
    val it = 46 : int
- it is the current value
ML Programs

- Programs are a sequence of statements
- Statements are:
  - bindings (like declarations of variables or functions)
  - type definitions
  - expressions to evaluate
- Some imperative flavor, but most everything has a value
  - All functions have values
  - No pure control flow
  - No assignment

ML Operators

- Usual arithmetic operators
  - Addition, subtraction, multiplication: + − ×
  - Integer division: div and floating point division: /
  - Note that int's and real's cannot be combined
- Unary negation denoted by tilde: ~ (not the minus sign)
- String concatenation with ^
- Comparison operators (work for numbers, bool, strings)
  - Usual <, <=, >, >=, =
  - Inequality is the two character operator <>
- Boolean operators
  - Logical and: andalso
  - Logical or: orelse
  - Logical negation: not
- Conditional expression: if expr1 then expr2 else expr3
Example Interpreter Evaluation

- 3 + 5;
val it = 8 : int
- 3 - 5;
val it = ~2 : int
- 3 div 5;
val it = 0 : int
- 5 div 3;
val it = 1 : int
- 5.0 / 3.0;
val it = 1.66666666667 : real
- 5 mod 3;
val it = 2 : int
- 5 <> 3;
val it = true : bool
- 5 = 3;
val it = false : bool

More Simple Examples

- "hello" ^ " " ^ "world";
val it = "hello world" : string
- "hello" < "world";
val it = true : bool
- "hello" > "world";
val it = false : bool
- 2 > 3 orelse true;
val it = true : bool
- 2 > 3 andalso true;
val it = false : bool
- 2 > 3 and true;
val it = true : bool

Error: syntax error found at AND
- not 2 > 3;
Error: operator and operand don't agree
- not (2>3);
val it = true : bool
- if true then 1 else 2;
val it = 1 : int

and is used for something else

= ;
Note this is an expression, not an imperative statement
ML Type Consistency

- ML has strong type checking
  - Types of operands to arithmetic operators must be same
    - 1 + 2;
      \texttt{val it = 3 : int}
    - 1.0 + 2.0;
      \texttt{val it = 3.0 : real}
    - 1 + 2.0;
      \texttt{Error: operator and operand don't agree}
  - No implicit type conversion (e.g., from int to real)
    - However, you can make a real out of an int with an explicit constructor \texttt{real}
    - You can convert a real to an int with several library functions: \texttt{floor ceil round trunc}

Conversion Examples

- \texttt{floor(3.5)};
  \texttt{val it = 3 : int}
- \texttt{ceil(3.5)};
  \texttt{val it = 4 : int}
- \texttt{round(3.5)};
  \texttt{val it = 4 : int}
- \texttt{trunc(3.5)};
  \texttt{val it = 3 : int}
- 1 + \texttt{round(3.5)};
  \texttt{val it = 5 : int}
- 3.5 + \texttt{real(it)};
  \texttt{val it = 8.5 : real}
- \texttt{int(it)};
  \texttt{Error: unbound variable or constructor: int}
### ML Value Names

- ML has named values (not really variables)
  - Identifier name is bound to a value
  - Like a declaration, initialization required (no assignment)
  - Identifiers must be defined and bound before use
  - Type may be specified, but is otherwise inferred from value

```
- val x = 13;
val x = 13 : int
- val z = x + 5;
val z = 18 : int
- val y : int = 2;
val y = 2 : int
- val y : real = 0;
  Error: pattern and expression in val dec don't agree
- val y : real = 0.0;
val y = 0.0 : real
```

### Type Constructors

- Create tuple values with list of values
  - Similar to struct in C, but no field names
  - Parentheses used for tuple construction
  - Access is positional with #n

```
- (2, 3);
val it = (2,3) : int * int
- ("if", true);
val it = ("if",true) : string * bool
- val city = ("Eugene", "OR", 97402);
val city = ("Eugene","OR",97402) : string * string * int
- val addr = ("123 Main", city);
val addr = ("123 Main",("Eugene","OR",97402)) : string * (string * string * int)
- val state = #2(city);
val state = "OR" : string
```
ML Lists

- Lists are homogeneous – values all same type
- Similar to arrays in C
- Brackets used to construct lists
- Functions \texttt{hd}, \texttt{tl} to access list

```ml
- \texttt{val L = ["first", "second", "third"];}
  \texttt{val L = ["first","second","third"] : string list}
- \texttt{hd(L);}
  \texttt{val it = "first" : string}
- \texttt{tl(L);}
  \texttt{val it = ["second","third"] : string list}
- \texttt{tl(tl(L));}
  \texttt{val it = ["third"] : string list}
- \texttt{hd(tl(tl(L)));
  val it = "third" : string}
```

ML List Operators

- \textit{Append} one list to another: operator \texttt{@}
- \textit{Prepend} an element to a list: operator \texttt{::}
  - Like cons in Scheme

```ml
- \texttt{val L@nil;}
  \texttt{val it = ["first","second","third"] : string list}
- \texttt{L@L;}
  \texttt{val it = ["first","second","third","first","second","third"] : string list}
- \texttt{val L2 = L::"fourth";}
  \texttt{Error: operator and operand don't agree}
- \texttt{val L2 = "zero":;L;}
  \texttt{val L2 = ["zero","first","second","third"] : string list}
```
Functions in ML

- ML is a functional language
  - A program is a function application
- Functions are defined with the keyword `fun`
  - Formal arguments are identifiers
  - Type of arguments may be inferred or explicit
  - Body of function is an expression

```ml
- fun f x = x + 1;
  val f = fn : int -> int
- fun g x = x + x;
  val g = fn : int -> int
- fun g x:int = x + x;
  val g = fn : int -> int
- fun h x = real x + 0.5;
  val h = fn : int -> real
```

Function Application

- Parentheses are not required by syntax
  - ML knows the type of everything
  - If a value's type is a function, and it is followed by anything, then the syntax indicates function application
  - Parentheses could be used for emphasis

```ml
- f 7;
  val it = 8 : int
- g 7;
  val it = 14 : int
- g(7);
  val it = 14 : int
- h 4;
  val it = 4.5 : real
```
Anonymous Functions

- Functions do not have to be bound to names
  - Use keyword `fn` and syntax `=>`
  - Similar to Lambda in Scheme
  - Functions are first class values

```ml
- val f = fn x => x + 5;
  val f = fn : int -> int
  - f 7;
  val it = 12 : int
  - (fn x => x+5) 7;
  val it = 12 : int
```

More on Function Application

- What about functions with two arguments?
  - If parentheses are used in definition then the function has a single argument which is a tuple of two values, so parentheses are also required in application

```ml
- fun f(x, y) = x + y;
  val f = fn : int * int -> int
  - f (3, 7);
  val it = 10 : int
  - f 7;
  Error: operator and operand don't agree
  - f 3 7;
  Error: operator and operand don't agree
```
Function Arguments

- What if we have two arguments and no parentheses?
  - Then we are actually defining a function that returns a function
  - The function has a single argument: \( x \)
  - The returned function also has a single argument: \( y \)
  - The expression is the definition of the returned function

\[
\text{fun } f \ x \ y = x + y;
\]

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

\[
\text{val } f 3 \ 7;
\]

\[
\text{val } \text{it} = 10 : \text{int}
\]

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

\[
\text{val } f (3,7);
\]

Error: operator and operand don't agree

A closer look at arguments

- All functions have exactly one argument
  - May be a tuple with many fields (or even none)
- Functions are first class values, so can be returned by functions
  - And bound to variables
- A function with multiple arguments (not a tuple) is called \textit{curried}
  - A function with multiple arguments as a \textit{tuple} is the \textit{uncurried} form

\[
\text{fun } f \ x1 \ x2 \ x3 = x1 + x2 + x3;
\]

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

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

\[
\text{val } g (1, 2, 3);
\]

\[
\text{val } \text{it} = 6 : \text{int}
\]

\[
\text{val } \text{it} = 6 : \text{int}
\]

\[
\text{val } f (1, 2, 3);
\]

Error: operator and operand don't agree

\[
\text{val } g 1 \ 2 \ 3;
\]

Error: operator and operand don't agree
More on Currying

- We may also define a curried function with anonymous notation

\[
\text{val } F = \text{fn } x1 \Rightarrow \text{fn } x2 \Rightarrow \text{fn } x3 \Rightarrow x1 + x2 + x3;
\]

\[
\text{val } F = \text{fn : int } \rightarrow \text{int } \rightarrow \text{int } \rightarrow \text{int}
\]

- \( F \ 1 \ 2 \ 3; \)

\[
\text{val } \text{it} = 6 : \text{int}
\]

- Providing fewer than all arguments gives function that is a partial instantiation of the function

\[
\text{val } F = 5;
\]

\[
\text{val } \text{it} = \text{fn : int } \rightarrow \text{int } \rightarrow \text{int}
\]

- \( \text{val } G = F 5; \)

\[
\text{val } G = \text{fn : int } \rightarrow \text{int } \rightarrow \text{int}
\]

- \( \text{val } H = G 13; \)

\[
\text{val } H = \text{fn : int } \rightarrow \text{int}
\]

- \( H \ 7; \)

\[
\text{val } \text{it} = 25 : \text{int}
\]

Patterns in Functions

- Consequence of strong type checking and inference
- Use to distinguish cases by type (or literal value)
  - Cases delineated by |
  - Wild card marked by _

\[
\text{fun } \text{fact } 0 = 1
\]

\[
| \text{fact } n = n * \text{fact } (n-1);
\]

\[
\text{val } f = \text{fn : int } \rightarrow \text{int}
\]

- \( \text{fact } 7; \)

\[
\text{val } \text{it} = 5040 : \text{int}
\]

- \( \text{fun } \text{fact } 0 = 1 \)

\[
| \text{fact } n = n * \text{fact } n-1;
\]
Patterns in Case Expressions

- Multiway branch, similar to switch in Java
  - But patterns are used in the cases
  - Function patterns are really syntactic sugar for case

```
- fun fact n = 
  case n of 
    0 => 1 
    | _ => n * fact(n-1);
val fact = fn : int -> int
- fact 10;
val it = 3628800 : int
```

Patterns with Lists

- List operators require element and list operands
  - We use this to form list patterns

```
- fun append([],L) = L 
  = | append(h::t,L) = h::append(t,L);
val append = fn : 'a list * 'a list -> 'a list
- append ([1,2,3], [4,5,6]);
val it = [1,2,3,4,5,6] : int list

- fun elt(1,x::xs) = x 
  = | elt(i, L) = elt(i-1, tl L);
val elt = fn : int * 'a list -> 'a
```
Recursion

- Reverse a list

```ml
fun reverse(nil) = nil
  | reverse(x::xs) = reverse(xs) @ [x];

val reverse = fn : 'a list -> 'a list
  | reverse([])
val it = [] : ?.X1 list
  | reverse([1,2,3,4])
val it = [4,3,2,1] : int list
  | reverse(reverse([1,2,3,4]))
val it = [1,2,3,4] : int list
```

- Rewritten with tail recursion

```ml
fun rev(nil, L) = L
  | rev(x::xs, L) = rev(xs, x::L);

val rev = fn : 'a list * 'a list -> 'a list
  | rev([1,2,3,4], [])
val it = [4,3,2,1] : int list
```

Non Linear Recursion

- Combinations of n things taken m at a time

```ml
fun comb(n,m) = fact(n) div (fact(m)*fact(n-m));

val comb = fn : int * int -> int
  | comb(4,2)
val it = 6 : int
  | comb(10,3)
val it = 120 : int
  | comb(20,5)
val it = 593775 : int

uncaught exception overflow
```

- Rewritten to be non-linear

```ml
fun comb(n,m) = if m=0 orelse m=n then 1
                 else comb(n-1,m) + comb(n-1,m-1);

val comb = fn : int * int -> int
  | comb(20,5)
val it = 15504 : int
  | comb(30,6)
val it = 593775 : int
```

Spring 2006
CIS 425 - ML
Mutual Recursion

- Two functions: odds gets elements 1, 3, 5,…
  - odds is just first element
  - plus evens of tail
- evens gets others
  - evens is just odds of tail

- fun odds(L) = if L = nil then nil else hd(L)::evens(tl(L))
  - Error: unbound variable or constructor: evens
- fun evens(L) = if L = nil then nil else odds(tl(L))
  - Error: unbound variable or constructor: odds

Solution: define both at once

- fun odds(L) = if L = nil then nil else hd(L)::evens(tl(L))
  - and evens(L) = if L = nil then nil else odds(tl(L))
  - val odds = fn : ''a list -> ''a list
  - val evens = fn : ''a list -> ''a list
  - odds([1,2,3,4,5])
  - evens([1,2,3,4,5])
  - val it = [2,4] : int list

Environments

- ML has an environment of current bindings
- So far, everything is in top level environment
- Similar to global variables in C/C++
- Local environment can be created with a let expression
  - Uses keywords let, in, end

- fun volume(r,h) =
  - let fun square(x:real) = x*x;
  - val pi = 3.14159;
  - in
  - pi * square(r) * h
  - end;
  - val it = 56.54862 : real
  - Error: unbound variable or constructor: pi
  - square(2.0);
  - Error: unbound variable or constructor: square
Scope

- This ML code works this way:

  ```ml
  let val x = 5;
  fun f y = x - y
  in
  let val x = 3
  in f x
  end
  end;
  val it = 2 : int
  ```

- What kind of scope does ML implement?

Defining Data Types

- In ML you can create user defined types
  - Similar to structures or classes
  - Uses the keyword `datatype`
- Here is a simple type that is like a C enumerated type

  ```ml
  datatype Color = Red | Yellow | Blue;
  datatype Color = Blue | Red | Yellow
  fun f Red = true
  | f Yellow = false
  | f Blue = true;
  val f = fn : Color -> bool
  - f Red;
  val it = true : bool
  - f Yellow;
  val it = false : bool
  ```
Data Constructors

- Constructors are ways to make new types out of values
  - Uses the keyword `of`
    - `datatype num = Int of int | Real of real;
      datatype num = Int of int | Real of real
    - `fun square (Int n) = Int(n*n)
      | square (Real x) = Real(x*x);
      val square = fn : num -> num
    - `square 2;
      Error: operator and operand don't agree
    - `square (Int 2);
      val it = Int 4 : num
    - `square (Real 3.0);
      val it = Real 9.0 : num

Why not just n*n and x*x ?

More Data Types

- Types can be recursively defined:
  - `datatype List = nil | cons of int * List;
    datatype List = nil | cons of int * List;
  - `val a = cons(1,cons(2,cons(3,nil)))
    val a = cons (1,cons (2,cons #)) : List
  - `fun sum(nil) = 0
    | sum(cons(n,L)) = n + sum(L);
    val sum = fn : List -> int
    - `sum a;
      val it = 6 : int

Does this function return two different types?
More Data Types

- Types can be unifying types and leave component types unspecified (polymorphic types)

  - `datatype ('a,'b) element = Pair of 'a * 'b | Single of 'a;
  datatype ('a,'b) element = Pair of 'a * 'b | Single of 'a`

  - `fun sum(nil) = 0 = | sum(Single(x)::L) = sum(L) = | sum(Pair(x,n)::L) = n + sum(L);`
  `val sum = fn : ('a,int) element list -> int`

- `val a = [ Pair("Bill",2), Single("Bob"), Pair("John",1), Single("Dan"), Pair("Joe",3) ];`

- `val a; val it = 6 : int`

Type Inference

- How does ML know types?
  - We don't usually specify types in value declarations or function definitions
  - ML uses Hindley-Milner type inference
    - Able to infer types from application of type consistency rules
    - Also uses knowledge of types of constants, type constructors, operators type requirements, etc.
  - With type inference, the programmer does not need to specify types, yet the language is still strongly typed
    - Some types are polymorphic – they are left unspecified and are instantiated to specific types when evaluation occur
Type Checking

- ML places various constraints on types
  - Types of operands to most operators must be the same – no conversions like in C and Java
  - ML assumes arithmetic involves int's unless the operands are explicitly real's
  - Elements of lists must all be the same type
  - A function always returns a single type
  - A function argument is always of the same type (no overloading)
- Not all types are equality types (comparable with =)
  - Functions cannot be compared for equality
  - Real numbers cannot be compared for equality (since they are approximations)

Inferring Types

- Start by assigning type place holders to each type in an expression
- Write down all relationships between types, using facts from type checking rules
- Use relationships to eliminate as many place holders as possible
- Use any type information (literals, operators, repeated value use, …) to determine concrete types
- Keep going until all types are known or we have a minimal number of unspecified types
Type Inference Example

- Suppose we have this function:
  \[
  \text{fun sum} \ [] = 0 \\
  \mid \text{sum} \ (x::xs) = x + \text{sum} \ xs;
  \]

- We can write down the following:
  - First, `sum` is a function, so has the form `A \to B`
  - From first clause body, we see that `B = \text{int}`
  - From second clause argument, we see that `A = A_1 \text{ list}`
  - From second clause body and `B=\text{int}`, we know that the addition must be between two int's, thus `x` must be an int, i.e., `A_1 = \text{int}`
  - We conclude that the function type is `(\text{int list} \to \text{int})`

- Verify in the ML interpreter:
  \[
  \text{val sum} = \text{fn} : \text{int list} \to \text{int}
  \]

Another Inference Example

- Suppose we have this function:
  \[
  \text{fun foo} (\text{nil, l}) = l \\
  \mid \text{foo} (x::xs, l) = x::\text{foo} (xs, l);
  \]

- We can write down the following:
  - First, `foo` is a function with two args, so has the form `A*B \to C`
  - From first clause body, we see that `B = C`
  - From second clause argument, we see that `A = A_1 \text{ list}`
  - From second clause body and operator we see that `B = B_1 \text{ list}`
  - From use of `x` in second clause body, we see that `A_1 = B_1`
  - We conclude that the type is `A_1 \text{ list} * A_1 \text{ list} \to A_1 \text{ list}`

- Verify in the ML interpreter:
  \[
  \text{val foo} = \text{fn} : 'a \text{ list} * 'a \text{ list} \to 'a \text{ list}
  \]
Another Example

- Suppose we have this function:
  
  ```ml
  fun bar(f, []) = []
  | bar(f, x::y) = (f x)::bar(f, y);
  ```

- We reason as follows
  - First, bar is a function with two args, so has the form $A*B \rightarrow C$
  - From second clause args, we see that $B = B1 \text{ list}$
  - From second clause body and function application, we know $f$ is a function with one arg, so $A = D \rightarrow E$
  - From use of $x$ in second clause, we see that $x$ has type $B1$, so $D = B1$
  - From second clause body, we know that $C = C1 \text{ list}$
  - From second clause body we also know that $E = C1$
  - We conclude that the type is $(B1 \rightarrow C1) * B1 \text{ list} \rightarrow C1 \text{ list}$

- Verify in the ML interpreter:
  ```ml
  val bar = fn : ('a -> 'b) * 'a list -> 'b list
  ```

Assignment in ML

- Assignment is the operator `:=`
  - Only works on references - typical val declarations are rvalues, not lvalues.
  - References are declared with `ref`, value obtained with operator `!`
    - `val x = ref 5;`
    - `val x = ref 5 : int ref`
    - `!x;`
    - `val it = 5 : int`
    - `x := !x + 1;`
    - `val it = () : unit`
    - `!x;`
    - `val it = 6 : int`
    - `x;`
    - `val it = ref 6 : int ref`
    - `x := x + 1;`
    - `Error: operator and operand don’t agree`
  - Using assignment means functions lose referential transparency