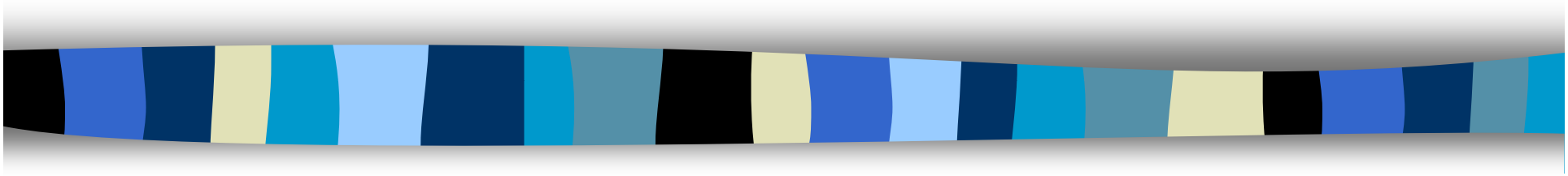
SML – A Functional Language



Lecture 19

Introduction to SML

- SML is a functional programming language and acronym for Standard Meta Language.
- SML has basic data objects as expressions, functions and list etc.
  - Function is the first class data object that may be passed as an argument, returned as a result and stored in a variable.
- SML is interactive in nature
  - Each data object is entered, analyzed, compiled and executed.
  - The value of the object is reported along with its type.
  - SML is **strongly typed language.**
  - Type can be determined automatically from its constituents by the interpreter for data object if not specified.
  - It is statically scoped language where the scope of a variable is determined at compile time that helps more efficient and modular program development.

Interaction with SML

- Basic form of interaction is
  - read, evaluate and display.
- An expression denotes a value and is entered and terminated by semi colon (;).
  - It is analyzed, compiled and executed by SML interpreter and
  - the result is printed on the terminal along with its type.
- In SML, the basic types are
  - *int* (integers), *real* (real), *char* (character),
  - *bool* (boolean) and *string* (sequence of character).

Conventions

- The following conventions are used to distinguish between user input and SML system's response.
- The SML editor prompts with " - " for an expression to be entered by an user.
- It displays the output after the symbol " > ".

```
-       2 + 5;                  <----  user's input
>       val it = 7 : int        <----  system's response
-       3 + 2.5;
>       Error: operator and operand don't agree
```

Cont…

- The result starts with the reserved word ***val***
- It indicates that the value has been computed and is assigned to system defined identifier named as ***it.***
- Its type is implicitly derived by the system from expression's constituents.
- Each time a new expression is entered, the value of ***it*** gets changed.

```
-       not false;
>       val it = true : bool
-       ~3.45 ;                 {~ is unary minus }
>       val it =  ~3.45 : real
-       (25 + 5) mod 2 ;    {mod  gives remainder}
>       val it =  0 : int
```

Value Declaration

- Value can be given a name called ***variable***.
- The value of a variable can not be updated and the life time of a variable is until it is redefined.
- The keyword ***val*** is used to define the value of a variable.
- The general form of a value declaration is:

  val var = exp

- It causes a variable ***var*** to be bound to the value of an expression ***exp*** .

```
-   val x = 3 + 5 * 2 ;
>   val  x =  13 : int
-   val y = x + 3;
>   val y =   16 : int
-   y + x ;        <-- without value declaration
>   val it =  29 : int
```

Bindings and Environments

- The name of a variable is formed by using alphanumeric characters [a – z, A – Z, numerals, underscore (_) and primes (‘)] and it must start with letter.
- The collection of bindings at any particular state is called an ***environment*** of that state.
- Execution of any declaration causes extension or change in the environment.
- The notation used for environment is not of SML program but our own notation to explain the meaning of SML programs.
- The execution of the value declaration

  - *val x = 3 + 5 * 2* creates the following environment

  env = [x |⇒ 13 : int]
- Each execution creates updated environment.

Examples

```
-          val x = 3 + 5 * 2;
>          val x =   13 : int
env1 = [x  |⇒  13 : int ]
-           val y = x + 3;
>          val y =   16 : int
env2 = [x  |⇒  13 : int, y  |⇒  16 : int]
-          y + x ;
>          val it =  29 : int
env3 = [x  |⇒  13 : int , y  |⇒  16 : int, it  ⇒  29 : int]
-          val  x = ~1.23E~8 ;
>          val x =   ~1.23E8 : real
env4 = [y  |⇒  16 : int, it  |⇒  29 : int, x  |⇒ –1.23*108  : real ]
```

Multiple Bindings

- Multiple variables can be bound simultaneously using key word ***and*** as a separator.
  - It is also called *simultaneous declaration*.
- A ***val*** declaration for simultaneous declaration is of the form

  val v1 = e1 and v2 = e2 and … and vn = en
- SML interpreter evaluates all the expressions e1, e2, .. en and then binds the variables v1, v2,… ,vn to have the corresponding values.
- Since the evaluations of expressions are done independently, the order is immaterial.

Examples: Contd…

- Continue with the previous environment

env4 = [y |⇒ 16 : int, it |⇒ 29 : int, x |⇒ –1.23*108 : real]

```
-        val y =  3.5   and   x = y ;
>        val y =  3.5 : real
>        val x = 16 : int
```

env5 = [it |⇒ 29 : int, y |⇒ 3.5 : real, x |⇒ 16 : int]

  - Note that x does not get the current value of y which is 3.5 but binds to the value 16 available from the previous environment *env4*
  - In multiple value bindings, the values on right hand sides are evaluated first and then bound to the corresponding variable in the left hand sides.

```
-        val y = y + 3.0 and  x = y ;
>        val y = 6.5 : real
>        val x = 3.5 : real
```

env6 = [it |⇒ 29 : int, y |⇒ 6.5 : real, x |⇒ 3.5 : real]

Compound Declarations

- Two or more declarations can be combined and separated by semicolon.
- The general form of *compound declaration* is

  D1; D2 ; …; Dn
- SML first evaluates the first declaration D1, produces an environment, then evaluates the second declaration D2 , updates the previous environment and proceeds further in the sequence.
- It must be noted that the subsequent declarations in sequential composition may override the identifiers declared in the left hand side declarations.

Examples

- Consider previous environment as

env6 = [it |⇒ 29 : int, y |⇒ 6.5 : real, x |⇒ 3.5 : real]
- val x = 34; val x = true **and** z = x ; val z = x;
> **val x = 34 : int**
env7 = [it |⇒ 29 : int, y |⇒ 6.5 : real, x |⇒ 34 : int]
> **val x = true : bool**
> **val z = 34 : int**
env8 = [it |⇒ 29 : int, y |⇒ 6.5: real, x |⇒ true: bool, z |⇒ 34 : int]
> **val z = true : bool**
env9 = [it |⇒ 29:int, y |⇒6.5: real, x |⇒ true: bool, z |⇒ true: bool]

Expressions and Precedence

- Expressions in SML are evaluated according to operator precedence.
  - The higher precedence means earlier evaluation.
  - Equal precedence operators are evaluated from left to right.
- Operators are of two kinds viz., infix operator and unary operator.
  - *Infix operator* is placed between two operands and is also called *dyadic operator*.
  - An *unary operator* is always written in front of an operand and has higher precedence than any infix operator.
  - It is also called *monadic operator*.
- In SML, infix minus is represented by - whereas unary minus represented by ~.

Conditional Expressions

- The general form of *conditional expression*

  if E then E1 else E2

  – The type of expressions E1 and E2 should be the same whereas the type of E is *bool*.

  – If E is true then E1 is the value of the conditional expression otherwise E2.

- The *condition* is formed using *arithmetic, relational, boolean* and *string* operators.

- Priority of operators is: arithmetic operators, relational operators followed by boolean / logical operators.

Arithmetic Operators

Integers: +, -, *, div, mod, abs, ~ (unary minus)

Real : +, -, *, /, sqrt, floor, sin, cos etc.

- Arithmetic operators +, -, and * are defined for both integers and reals & overloaded.
- The operators are overloaded if defined for more than one type of data types.
- SML can deduce the type in most of the expressions, functions from the type of the constituents used.

Relational & Boolean operators

Integers & reals:

- < (less than),
- <= (less or equal to),
- > (greater than),
- >= (greater or equal to)

For all except reals

- = (equal to),
- <> (not equal to)

- Precedence is in decreasing order of *not, andalso* and *orelse*
  1. *not* (Logical negation),
  2. *andalso* (Logical AND),
  3. *orelse* (Logical OR)
- The boolean operators *andalso* and *orelse* are evaluated using lazy evaluation strategy which means that evaluate whenever it is required.

Boolean Operators – Cont…

- **andalso:** true only when both operands are true.
- **orelse**: false only when both operands are false.

```
-       val x = true andalso false;
>       val x = false : bool
-       val y = x orelse true;
>       val y = true : bool
-       val z = not true;
>       val z = false : bool
```

Cont…

```
-       val p = x orelse not(y);
>       val p = false : bool
-       val n = 5;
>       val n = 5 : int
-       val t = if n+3 > 0 orelse p then 9 else 6;
>       val t = 9 : int
-       val t = if p orelse not false then n else 3;
>       val t = 5 : int
```

Function Declaration

- Functions are also values in SML and are defined in the same way as in mathematics.
- A function declaration is a form of value declaration and so SML prints as the value and its type.
- The general form of function definition is:

  fun fun_name (argument_list) = expression

- The keyword "**fun**" indicates that function is defined.
- **fun_name** is user defined variable and argument_list consists of arguments separated by comma.

Function – Cont…

- Let us write a function for calculating circumference of a circle with radius **r**.

```
-       val   pi  =  3.1414;
>       pi = 3.1414 : real
-       fun  circum  ( r ) = 2.0 * pi * r;
>       val  circum  = fn : real  → real
-       circum (3.0);
>       val it = 18.8484 : real
```

  - Here "**circum**" is a function name.

Cont…

- SML can infer the type of an argument from an expression (2.0 * pi * r).
- Variables appearing in the argument list are said to be **bound** variables.
  - In the function "**circum**", **pi** is a free identifier whereas **r** is bound.
- If there is one argument of a function, then circular brackets can be removed.
- The same function can be written as:

```
-    fun  circum   r  = 2.0 * pi * r;
>    val  circum  = fn : real  → real
-    circum 1.5;
>    val  it = 9.4242 : real
```

Static Binding of Function

```
-       val pi = 1.0;
>       val pi = 1.0 : real
-       circum 1.5;
>       val  it = 9.4242 : real
```

- Note that the value of **circum 1.5** is still 9.4242 even though pi is bound to new value to 1.0.
- SML uses the environment valid at the time of declaration of function rather than the one available at the time of function application.
- This is called the **static binding** of free variables in the function.

Polymorphic Function Declarations

- Sometimes type of the function is not deducible seeing the arguments or the body of the function.
- These arguments can be of any type and called **polytype.**
- The actual type would be decided at the time of applying function.
- Function using polytype is called **polymorphic function**.

```
-        fun pair_self  x = (x, x) ;
>        val pair_self  =  fn :  'a  → 'a *  'a
```

  - Here ‘a denotes polytype.

Examples

```
-   val p = pair_self  25;
>   val p = (25, 25) : int * int
-   val p2 = pair_self  true;
>   val p2 = (true, true): bool*bool
-   fun first_of_pair  (x, y) = x ;
>   val first_of_pair  =  fn :  'a * 'b → 'a
-   val f = first_of_pair  (23, 4.5);
>   val f = 23 : int
-   fun second_of_pair  (x, y) = y ;
>   val second_of_pair  =  fn :  'a * 'b → 'b
-   val f = second_of_pair  (23, true);
>   val f = true: bool
```

Patterns

- A **pattern** is an expression consisting of variables, constructors and wildcards.
- The **constructors** comprise of **constants** (integer, character, bool and string), **tuples**, record formation, datatype constructors (explained later) etc.
- The simplest form of pattern matching is

  pattern = exp, where exp is an expression.
- When **pattern = exp** is evaluated, it gives true or false value depending upon whether pattern matches with an expression or not.

```
-       true = ( 2 < 3);
>       val it = true : bool
-       23 = 10 + 14;
>       val it = false : bool
```

Contd…

```
-       "abcdefg" = "abc" ^ "defg";
>       val it = true : bool
-       #"a" = # "c";
>       val it = false : bool
-       (23, true) = (10+13, 2< 3);
>       val it = true : bool
-       val v = 3;
>       val v = 3 : int
-       v = 2 + 1;
>       val it = true : bool
```

Contd…

- In SML, the pattern matching occurs in several contexts.
- Pattern in value declaration has the form

  val pat = exp
- If pattern is a simple variable, then it is same as value declaration.
- If patterns are Pairs, tuples, record structure, then they may be decomposed into their constituent parts using pattern matching.
- The result of matching changes the environment.
- Reduction to atomic value bindings is achieved where an atomic bindings is one whose pattern is a variable pattern.
- The binding **val (pat1, pat2) = (val1, val2)** reduces to

  > **val pat1 = val1**

  > **val pat2 = val2**

Contd…

- This decomposition is repeated until all bindings are atomic.

```
-       val ((p1, p2), (p3, p4 , p5)) = ((1,2), (3.4, “testing”,true));
>       val p1 = 1 : int
>       val p2 = 2 : int
>       val p3 = 3.4 : real
>       val p4 = “testing” : string
>       val p5 = true : bool
-       val (p1, p2, _ , p4) =(12, 3.4, true, 67);

>       val p1 = 12 : int
>       val p2 = 3.4 : real          wildcard
>       val p4 = 67 : int
```

- The wildcard pattern can match to any data object. Represented by underscore (_) and has no name thus returns an empty environment.

Alternative Pattern

- Functions can be defined using alternative patterns as follows:

  fun pat1 = exp1 | pat2 = exp2 | … | patn = expn ;

- Each pattern **patk** consists of same function name followed by arguments.
- The patterns are matched from top to bottom until the match is found.
- The corresponding expression is evaluated and the value is returned.

```
-       fun     fact 1 = 1
                | fact n = n * fact (n-1);
>       val fact = fn : int -> int
```

Contd…

```
>       val fact = fn : int -> int
-       fact 3;
>       val it = 6 : int
-       fun     negation true = false
                | negation false = true;
>       val negation = fn : bool -> bool
-       negation (2 > 3);
>       val it = true : bool
```

Function using Alternative

- Functions can be defined using alternative as follows:

```
fun     fun_def1 = exp1
        | fun_def2 = exp2

        | …

        | fun_defn = expn ;
```

  - Each definition is matched from top to bottom until the match is found.
  - The corresponding expression is evaluated and the value is returned.

Examples

```
-       fun     fact 1 = 1
                | fact n = n * fact (n-1);
>       val fact = fn : int -> int
-       fact 3;
>       val it = 6 : int
-       fun     negation true = false
                | negation false = true;
>       val negation = fn : bool -> bool
-       negation (2 > 3);
>       val it = true : bool
```

Case Expression

- Conditional expression takes care of only two cases whereas if we want to express more than two cases, then the nested **if-then-else** expression is used.
- Alternatively we can handle such situations using **case expression.**
- The general form of case expression is:

```
case exp of   pat1 => exp1
            | pat2 => exp2
                  .
                  .
            | patn => expn
```

Pattern – Cont…

- The value of **exp** is matched successively against the patterns **pat1 , pat2 , … , patn** .
- If **patj** is the first pattern matched , then the corresponding **expj** is the value of the entire case expression.
- For example the nested nested if-then-else expression

if x = 0 then "zero" else if x = 1 then "one" else if x = 2 then "two" else "none"

is equivalent to the following case expression

```
case x of        0 => "zero"
               | 1 => "one"
               | 2 => "two"
wild card      | _ => "none"
```

Lists in SML

- List is an ordered sequence of data objects, all of which are of the same type.
- In SML, the list consists of finite sequence of values of type ‘a and the entire list is of type ‘a ***list***.
- The elements of list are enclosed in square brackets and are separated by comma.
- The empty list denoted by [ ] that contains no elements.
- The order of elements is significant.

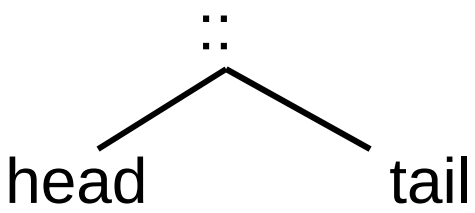
List – Cont…

- List can contain varying number of elements of the same type whereas in tuples and records the number of elements are fixed and are of any type.
- The first element of the list is at 0 position.
- Typical lists are:

```
-       val x = [];                    ← empty list
>       val x = [ ] : 'a list
-       val r = [2.3, 4.5 / 1.2, 8.9 + 2.3];← list of real
>       val r = [2.3,3.75,11.2] : real list
-       val y = [[1,2], [3,4,5,6], [ ]]; ← list of list of int type
>       val y = [[1,2],[3,4,5,6],[]] : int list list
-       val p = [floor, round, trunc];    ← list of functions
>       val p = [fn,fn,fn] : (real -> int) list
```

Construction of a List

- A list is constructed by two primitives: one a constant **nil** (empty list denoted by [ ]) and other an infix operator **cons** represented by ::
- A list is represented as **head :: tail** , where head is a first element of the list and tail is the remaining list.
- The operator cons builds a tree for a list from its head to tail.
- For example, a list [2] can be represented as **2 :: nil**
- The tree representation for a list **head :: tail** is as follows:



Cont…

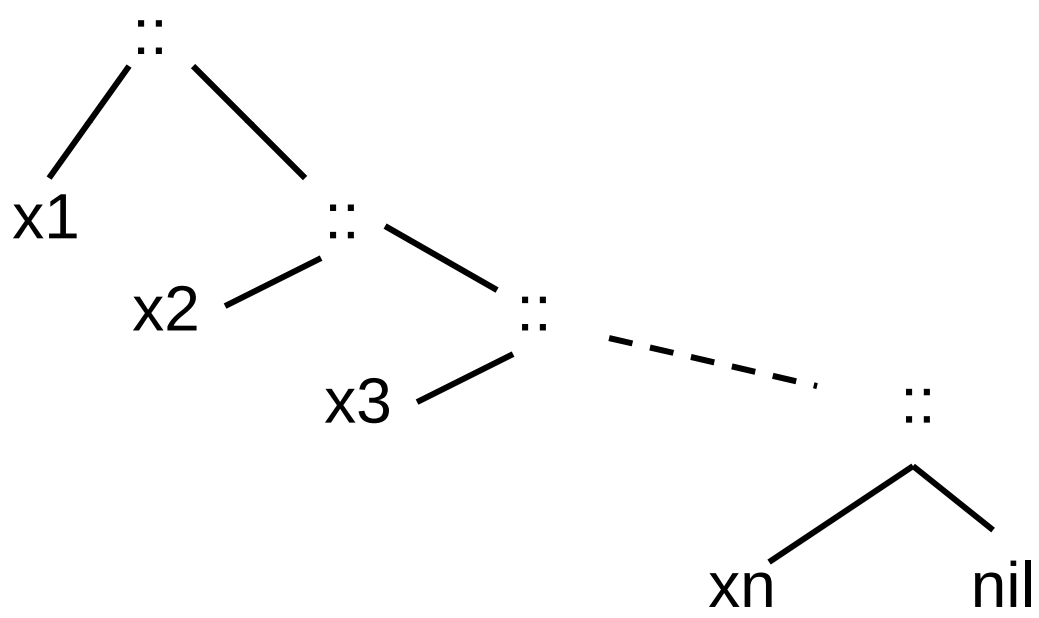
- A list can be constructed by adding an element in the beginning using cons operator.

```
-       4::nil;
>       val it = [4] : int list
-       val p = 3 :: [4];
>       val p = [3,4] : int list
-       val q = 2 :: p;
>       val q = [2,3,4] : int list
```

- The list [x1, x2, x3, …, xn] can also be written as
  x1 :: :: xn :: nil.

Cont…

- It’s tree representation is



- Two lists can be compared for equality or inequality.

Standard List Functions

- There are only few standard functions for handling lists in SML
- *List constructor operator denoted by* **::**

```
-       2 :: [3,4,5];
>       val it = [2,3,4,5] : int list
-       true :: [2>3];
>       val it = [true, false] : bool list
```

- *The Append operator denoted by @*

```
-       [1,2,3] @ [4,5];
>       val it = [1,2,3,4,5] : int list
```

List Functions

- *The reversing function rev for reversing the elements of a list*:

```
-       rev [1,2,3];
>       val it = [3,2,1] : int list
-       rev [[1,2], [3,4], [5]];
>       val it = [[5],[3,4], [1,2]] : int list list
-       rev [(10,"abc"), (20, "bcd")];
>       val it = [(20,"bcd"),(10,"abc")] : (int * string) list
```

List Functions – Cont…

- *Finding Head and Tail of a list by using **hd** (for head) & **tl** (for tail) functions*:

```
-       hd [(1,"ab"), (2, "bc")];
>       val it = (1,"ab") : int * string
-       tl  [(1,"ab"), (2, "bc")];
>       val it = [(2,"bc")] : (int * string) list
```

- *Finding the length of a list length*:

```
-       length [1,2,3,6];
>       val it = 4 : int
-       length [(1,"ab"), (2, "bc")];
>       val it = 2 : int
```

Various other list functions

- Adding elements of the list

```
-       fun     add []    =       0
                | add (x::xs)     =       x + add xs;
>       val add  =  fn : int list -> int
-       add [2,3,4,7,8];
>       val it = 24 : int
```

- Multiplying elements of the list

```
-       fun     mult  []      =       0
                | mult [x]    =       x
                | mult (x::xs) =      x * mult xs ;
>       val multl = fn : int list -> int
```

Cont…

- Selecting a particular position value

```
-        fun  select (n, [])   =          0
                | select (n, (x::xs)) = if n = 1 then x
                                        else select ((n-1), xs);
>        val select = fn : int * int list -> int
```

- Finding the maximum value of the elements in a list

```
-        fun    max []        =          0
               | max [x]      =          x
               | max(x::y::xs) = if x > y then
                                 max(x::xs) else max(y::xs);
>        val max = fn : int list -> int
-        max [3,9,1,3,56,7];
>        val it = 56 : int
```