Expressions in SML

• Computation is evaluation of expressions.
• Every expression...
  1. ...has a type
     • e.g., int or bool or int*string->int
  2. ...may result in a value when evaluated
     • e.g., 7 or "hello" or fn x => x+1
  3. ...may cause side-effects when evaluated
     • e.g., assignments, input/output, exceptions
Types and Values

• If \( \text{exp} \) is an expression and \( \text{ty} \) is a type, we use the notation
  
  \[
  \langle \text{expression} \rangle : \langle \text{type} \rangle
  \]
  
  to mean that the given expression has the given type.

• A value is an expression which evaluates to itself.

Base Types and Values

\[
3 : \text{int} \\
17 : \text{int} \\
\sim 4 : \text{int}
\]

true : bool
false : bool
More Base Types and Values

3.14 : real
2.17 : real
6.022e-23 : real
"hello world\n" : string
#"a" : char
#"\n" : char
() : unit

Simple Expressions

• What are the types and values of the following expressions?

3+4
~3 - ~4
3.14 <= 2.17
if (7<2) then "yes" else "no"
"hello " ^ "world\n"
Pairs

- Values
  - (3, true) : int*bool
  - (~17, false) : int*bool
  - ("pi", 3.14) : string*real
  - (~12, 4) : int*int

- Expressions: left to right evaluation
  - (3+9, ~(~4)) : int*int

Tuples

- (4, "cs131", 6-2) : int*string*int
  - and so on, for as many components as you want

- Careful... the following types are all different!
  - int * string * int
  - (int * string) * int
  - int * (string * int)
Lists

• Lists are defined inductively:
  - The constant nil is a list (the empty list).
  - If h is an element and t is a list then
    \[ h :: t \]
    is a list whose first element is h and whose remaining elements are the elements of t.

• Typechecking: lists must be homogenous
  - All elements in a list must have the same type.
  - List types written \texttt{int list} and \texttt{bool list} and \texttt{(int*int) list}

List Examples

\[
\begin{align*}
\text{nil} & : \text{int list} \\
2::\text{nil} & : \text{int list} \\
1::(2::\text{nil}) & : \text{int list}
\end{align*}
\]

\[
\begin{align*}
\text{nil} & : \text{bool list} \\
("x"="y")::\text{nil} & : \text{bool list} \\
(1\lt2)::(("x"="y")::\text{nil}) & : \text{bool list}
\end{align*}
\]
Alternate List Notation

\[
\begin{align*}
nil &= [] \\
3::nil &= [3] \\
2::(3::nil) &= [2,3] \\
1::(2::(3::nil)) &= [1,2,3] \\
(1<2)::(("x"="y")::nil) &= [1<2,"x"="y"] \\
[[1,2],[3,4,5]] : (int list) list
\end{align*}
\]

Variable Bindings

- General form:
  \[
  \text{val} \ <\text{pattern}> = \ <\text{expression}>
  \]
  One possible pattern is simply a variable
  \[
  \begin{align*}
  \text{val} \ x &= 3 + 4 \\
  \text{val} \ x' &= x + 1 \\
  \text{val} \ s1 &= \text{"foo"} ^ \text{"bar"} \\
  \text{val} \ lst &= \{1+2, \ 3+4\} \\
  \text{val} \ \text{Long\_Variable\_Name} &= \{\text{lst, lst}\}
  \end{align*}
  \]
Variable Bindings

• Evaluation of the definition

\[
\text{val } x = \text{<expression>}
\]

proceeds by evaluating the expression, and binding the resulting value to the new variable \( x \).

• This is not an assignment statement, but a new variable declaration.
  - In fact, "variables in SML cannot be assigned to".

Patterns

• Wildcard Pattern: throws away the result

\[
\text{val } _\_ = \text{print } "\text{hello world}\text{"}
\]

• Constant Pattern: just checks for a match

\[
\text{val } 7 = 2+5
\]

• Tuple Pattern: matches component-wise

\[
\text{val } (x,y) = (3,4) \\
\text{val } (w,(x',\_),z) = (\text{true},(y,x),x)
\]

• List Pattern: matches against head and tail

\[
\text{val } (x::y) = [3,4,5] \\
\text{val } ((\text{a},3)::c) = [(x,x)]
\]
Patterns

• Restriction: all patterns in SML must be linear
  - That is, variables cannot be repeated.
  - Can't have definitions like
    \[
    \text{val } (x,y,x,\_)= (3,4,3,\text{true})
    \]
    even though this should "intuitively" match the pattern.

Function Values

• How do we write the successor function on integers, that adds one to its argument?
  \[
  \text{fn } (x: \text{int}) \Rightarrow x+1
  \]

• Q: What is the name of this function?
  • A: It doesn't have a name! It's just a thing that happens to map inputs to outputs as shown.
Function Values

• How do we write the successor function on integers, that adds one to its argument?
\[
\text{fn } (x:\text{int}) \Rightarrow x+1
\]
• Q: What is the type of this function?
• A: It takes an integer argument and returns an integer result, so its type is
\[
\text{int } \rightarrow \text{ int}
\]

Function Values

• How do we give a name to this function?
\[
\text{val succ } = \text{ fn } (x:\text{int}) \Rightarrow x+1
\]
• How do we apply this function?
\[
\text{succ}(3)
\]
\[
\text{succ } 3
\]
\[
(\text{fn } (x:\text{int}) \Rightarrow x+1) \ 3
\]
• Careful: \text{succ}(3*2) is not \text{ succ } 3 \times 2
  - What is the value of \text{ succ } 3 \times 2 ?
Defining Factorial

• What's wrong with the following definition?

```scala
val factorial = 
  (fn n => if (n=0) then
    1
  else
    n*factorial(n-1))
```
Defining Factorial

```ml
val rec factorial = 
  (fn n => if (n=0) then 1 else n*factorial(n-1))
```

• Then
  
  factorial : int -> int
  factorial 3 : int

Better Syntax For Functions

```ml
fun succ(n:int) = n+1

fun factorial(n:int) = 
  if (n=0) then 1
  else n*factorial(n-1)
```
Better Syntax For Functions

• SML can (almost always) infer the types of variables for you, so they can usually be omitted if desired.

fun succ n = n+1

fun factorial n = 
    if (n=0) then 
        1 
    else 
        n*factorial(n-1)

Pattern-Matching in Functions

• Functions can be defined by pattern-matching

fun factorial 0 = 0 
    | factorial n = n * factorial(n-1)

• Evaluation rule: pick the first clause that matches the argument value.
Pattern-Matching In Functions

• Function to multiply a list of integers

\[
\text{fun} \quad \text{prod} \; [\;] = 1 \\
| \quad \text{prod} \; (n::ns) = n \ast (\text{prod} \; ns)
\]

• What is the type of \text{prod}?

Pattern-Matching In Functions

• Function to multiply a list of integers

\[
\text{fun} \quad \text{prod} \; [\;] = 1 \\
| \quad \text{prod} \; [n] = n \\
| \quad \text{prod} \; (n::ns) = n \ast (\text{prod} \; ns)
\]

• Which patterns would the list \([3]\) match?
Multi-argument Functions

• Every function takes exactly one argument, but that argument might be a tuple (or a record)

```latex
fun power(x, n) = 
    if (n = 0) then 
        1.0 
    else 
        x * power(x, n-1)

power(2,3) : int
```

Multi-argument Functions

• Alternate definition

```latex
fun power(x, 0) = 1.0
    | power(x, n) = x * power(x, n-1)

power(2,3) : int
```
Let-Expressions

• Method of local variable declarations
• Have the form
  \[ \text{let } \langle \text{definitions} \rangle \ \text{in} \ \langle \text{expression} \rangle \ \text{end} \]
• Evaluation process:
  - Evaluate definitions in sequence, binding any variables
  - Evaluate the expression (the "body" of the let)
  - Forget the new variable bindings
  - Return the value of the body

Let-Expression Example

fun solve_quadratic(a,b,c) =
  let
    val disc = b*b - 4.0*a*c
    val sqrdisc = Math.sqrt disc
    val denom = 2.0*a
  in
    (~~b + sqrdisc) / denom,
    (~~b - sqrdisc) / denom
  end
Length of a List

• Length function for integer lists:
  \[
  \text{fun} \quad \text{length} \; (\; : \text{int list}) = 0 \\
  \quad | \quad \text{length} \; (\; :: \text{xs}) = 1 + \text{length} \; \text{xs}
  \]

• Better definition:
  \[
  \text{fun} \quad \text{length} \; (\; : \text{int list}) = 0 \\
  \quad | \quad \text{length} \; (\; :: \text{xs}) = 1 + \text{length} \; \text{xs}
  \]

• What is the type of \text{length} ?

Length of a List

• Length function without type annotation:
  \[
  \text{fun} \quad \text{length} \; [\; ] = 0 \\
  \quad | \quad \text{length} \; (\; :: \text{xs}) = 1 + \text{length} \; \text{xs}
  \]

• Now what should the type of \text{length} be?