Computer Science 131
Programming Languages

January 17, 2001

SML Introduction

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 `exp` is an expression and `ty` is a type, we use the notation

  `<expression> : <type>`

  to mean that the given expression has the given type.

- A *value* is an expression which evaluates to itself.

Base Types and Values

```
3  : int
17 : int
~4 : 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 `int list` and `bool list` and `(int*int) list`

List Examples

```
          nil   :   int list
        2::nil  :   int list
   1::(2::nil)  :   int list

                 nil  : bool list
      ("x"="y")::nil) : bool list
(1<2)::(("x"="y")::nil) : bool list
```

Alternate List Notation

```
            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
```

Variable Bindings

- General form:

  val <pattern> = <expression>

  One possible pattern is simply a variable

```
val x   = 3 + 4
val x'  = x + 1
val s1  = "foo" ^ "bar"
val lst = [1+2, 3+4]
val Long_Variable_Name = [lst,lst]
```

Variable Bindings

- Evaluation of the definition

  ```
  val x = <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

  ```
  val _ = print "hello world\n"
  ```
- Constant Pattern: just checks for a match

  ```
  val 7 = 2+5
  ```
- Tuple Pattern: matches component-wise

  ```
  val (x,y) = (3,4)
  val (w,(x',_),z) = (true,(y,x),x)
  ```
- List Pattern: matches against head and tail

  ```
  val (x::y) = [3,4,5]
  val ((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

```
val (x,y,x,_) = (3,4,3,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?

```
fn (x:int) => 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?

```
fn (x:int) => 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

```
int -> int
```

Function Values

- How do we give a name to this function?

```
val succ = fn (x:int) => x+1
```

- How do we apply this function?

```
succ(3)
succ 3
(fn (x:int)=>x+1) 3
```

- Careful: `succ(3*2)` is not `(succ 3)*2`
  - What is the value of `succ 3 * 2` ?

Defining Factorial

- What's wrong with the following definition?

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

Defining Factorial

```
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

```
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

```
fun prod [] = 1
  | prod (n::ns) = n * (prod ns)
```

- What is the type of `prod` ?

Pattern-Matching In Functions

- Function to multiply a list of integers

```
fun prod [] = 1
  | prod [n] = n
  | prod (n::ns) = n * (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)

```
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

```
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

  let <definitions> **in** <expression> **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 sqrtdisc = Math.sqrt disc
     val denom = 2.0*a
  in
     ((~b + sqrtdisc) / denom,
      (~b – sqrtdisc) / denom)
  end
```

Length of a List

- Length function for integer lists:

```
fun length ([] : int list) = 0
  | length (x::xs) = 1 + length xs
```

- Better definition:

```
fun length ([] : int list) = 0
  | length (_::xs) = 1 + length xs
```

- What is the type of `length` ?

Length of a List

- Length function without type annotation:

```
fun length []      = 0
  | length (_::xs) = 1 + length xs
```

- Now what should the type of `length` be?