F28PL1 Programming Languages

Lecture 12: Standard ML 2

Declaration

introduce a variable

associate identifier with value

- val *identifier* = *expression*;

> val *identifier* = *value* : *type*

identifier - any sequence of letters, digits and _

starting with a letter

use *identifier* in subsequent expressions to return *value*

Declaration

```
- val toPound = 0.66;
> val toPound = 0.66 : real
- 27.45*toPound;
> 18.117 : real
- val employee =
("Duncan","Thaw",19000.0);
> val employee =              ("Duncan","Thaw",19000.0):
                string * string * real
- val payPounds = #3 employee*toPound;
> val payPounds = 12540.0 : real
```

Declaration

can't assign new value to declared variable

can *redeclare* variable by entering new definition

```
- val toPound = 0.68;
> val toPound = 0.68 : real
```

creates a *new instance* of the variable

does not assign to existing variable

Function

function

mapping from a *domain* to a *range*

domain - formal parameters

range - result

function type displayed as

`fn : domain type -> range type`

Function

can display types of prefix system functions

```
- size;
> fn : string -> int
```

i.e. size takes a string and returns an int

```
- not;
> fn : bool -> bool
```

i.e. not takes a bool and returns an int

Function definition

fun *identifier pattern* = *expression*;

identifier

name associated with function

pattern

formal parameters

to begin with, an *identifier*

expression

function *body*

Function definition

system shows function type

can't display function value

```
- fun ident pattern = expression;
> val ident = fn : type1 -> type2
type1 - type for pattern - domain
type2 - type for expression - range
```

Function definition

system uses *polymorphic type checking* to deduce types from use of operators/values of known type

e.g. increment an integer

```
- fun inc x = x+1;

> val inc = fn : int -> int

1 is int so ...

+ must be int addition so...
```

- `x must be int`

Function definition

e.g. put "s" on end of a string

```
- fun plural w = w^"s";
> val plural = fn : string -> string
```

^ takes 2 strings so...

w must be string

^ returns a string so...

w must be string and "s" must be string

Function definition

e.g. given string, return it with its length

```
- fun sSize s = (s,size s);
> val sSize =
    fn : string -> string * int
size takes a string so...
 s must be string
size returns int
```

Explicit formal parameter types

sometimes can't deduce type in presence of overloaded operators

e.g. square an integer

```
fun sq x = x*x
```

can't tell if * is int or real

use *explicitly typed* pattern: (*pattern*:*type*)

```
- fun sq (x:int) = x*x;
> val sq = fn : int -> int
```

Function call

must call a *type1* -> *type2* function with a *type1* actual parameter to return a *type2* result

- *ident expression1*;

> *value2* : *type2*

system:

1. evaluates *expression1* to give *value1* of type *type1*
2. finds *type1* -> *type2* function associated with *ident*
3. replaces's *idents* formal parameter with *value1* in *idents* body
4. evaluates *idents* body to return *value2* of *type2*

Function call

substitution model of function call

```
- inc 3;
> 4 : int
inc 3 ==> 3+1 ==> 4
- plural “banana”;
> “bananas” : string
plural “banana” ==>
“banana”^”s” ==>
“bananas”
```

Function call

```
- sSize “fish”;
> (“fish”,4) : string * int
sSize “fish” ==>
(“fish”,size “fish”) ==>
(“fish”,4)
- sq 7;
> 49 : int
sq 7 ==> 7*7 ==> 49
```

Multiple arguments

```
fun ident patt1 patt2 ... pattN =
exp;

if:

patt1  : type1 ... pattN : typeN

exp    : typeN+1

then function type is:

type1 -> ... -> typeN -> typeN+1
```

Multiple arguments

e.g. sum of squares

```
- fun sumsq x y = sq x+sq y;
> val sumsq = fn : int -> int -> int
```

sq is int -> int so...

x must be int

sq is int -> int so...

y must be int

sq x is int and sq y is int so...

+ must be int addition

Multiple arguments

```
- sumsq 3 4;
> 25 : int
sumsq 3 4 ==>
sq 3+sq 4 ==>
9+16 ==>
25
```

Multiple arguments

e.g. does a string have more than n letters?

```
- fun isMore s n = size s > n;
> val isMore : fn string -> int -> bool
```

size takes a string so...

 s is string

size s returns an int so...

 > is int comparison so...

- n is int

> returns a bool

Multiple arguments

```
isMore “hello” 6;

> false : bool

ismore “hello” 6 ==>

size “hello” > 6 ==>

5 > 6 ==>

false
```

Multiple cases

can define functions with multiple cases

```
fun ident patterns1 = expression1 |
    ident patterns2 = expression2 |
  ...
```

patterns is one or more *pattern*

pattern may be

constant

identifier

patterns must be of same types

expressions must be of same type

Multiple cases

for function call:

actual parameters matched against each *patterns* in turn

when all match succeeds, value of corresponding *expression* returned

match:

constant - must be same as actual parameter value

identifier - *identifier* in body *expression* replaced with actual parameter

should have a case for all possible combinations of *patterns* values

Multiple cases

e.g. is integer 0?

```
- fun isZero 0 = true |
      isZero x = false;
> val isZero = fn : int -> bool
```

0 is int so...

x must be int

true and false both bool so...

result is bool

Multiple cases

```
- fun isZero 0 = true |
      isZero x = false;
> val isZero = fn : int -> bool
- isZero 0;
> true : bool
```

0 matches 0 ==> true

```
- isZero 3;
> false : bool
```

3 doesn’t match 0

3 matches x ==> false

Multiple cases

e.g. rainbow colour sequence

```
- fun next "red" = "orange" |
      next "orange" = "yellow" |
      next "yellow" = "green" |
      next "green" = "blue" |
      next "blue" = "indigo" |
      next "indigo" = "violet" |
      next s = s^": not a valid colour";
> val next = fn : string -> string
"red", "orange" ... all string so s must be string
all results are string
```

Multiple cases

```
- next “blue”;
> “indigo” : string
“blue” doesn’t match “red”
“blue” doesn’t match “orange”
...
“blue” matches “blue” ==> “indigo”
```

Multiple cases

```
- next “banana”;
> “banana not a valid colour” : string
“banana” doesn’t match “red”
...
“banana” matches s ==>
“banana”^” not a valid colour” ==>
“banana: not a valid colour”
```

Wildcard pattern

_ - matches anything and ignores it

e.g. don't use x on right hand side of 2nd case of iszero

```
- fun isZero 0 = true |
      isZero _ = false;
> val isZero = fn : int -> bool
- isZero 22;
> false : bool
```

22 doesn't match 0

22 matches _ ==> false

Non-sequential conjunction

```
- fun  AND false false = false |
       AND false true = false |
       AND true false = false |
       AND true true = true;
> val AND = fn : bool -> bool -> bool
- AND true false;
> false : bool
true false doesn't match false false
true false doesn't match false true
true false matches true false ==> false
```

Non-sequential conjunction

if 1st actual parameter is `false then result is false`

`if 1st actual parameter is true then result is same as second actual parameter`

`simplify:`

```
- fun AND false _ = false |
      AND true y = y;
> val it AND = fn : bool -> bool -> bool
```

Recursion 1

recursion is functional programming equivalent of iteration in imperative language

usually recurse across a range

base case

at end of range

function returns a value

*recursion cas*e

function combines result of processing next in range with result of processing rest of range

i.e. recursion case makes progress towards base case

Recursion 1

for integers, range is often from some positive value, say `n,  to 0`

```
base case: n=0 ==> return value

recursion case: n>0 ==> process n and
recurse with n-1

e.g. sum integers from 0 to n

base case: n=0 ==> sum is 0

recursion case: n>0 ==>

      add n to summing from 0 to n-1
```

Recursion 1: integer

```
- fun sum 0 = 0 |
      sum n = n+sum (n-1);
> val sum = fn : int -> int
0 is int so n must be int
0 is int so ...
  n+sum (n-1) must be int so ...
  + must be int addition so ...
  n must be int (!) and sum (n-1) must be
int
```

Recursion 1: integer

```
- sum 3;
> 6 : int
sum 3 ==>
3+sum 2 ==>
3+(2+sum 1) ==>
3 + (2+(1+sum 0)) ==>
3+(2+(1+0)) ==>
6
```

Recursion 1: integer

e.g. find 2 to the power `n`

```
base case: n=0 ==> 1

recursion case: n>0 ==> multiply 2 by 2n-1

- fun pow2 0 = 1 |
      pow2 n = 2*pow2 (n-1);
> val pow2 = fn : int -> int

0 is int so n must be int

0 is int so * must be int multiplication so
2 must be int  and pow2 (n-1) must be int
```

Recursion 1: integer

```
- pow2 3;
> 8 : int
pow2 3 ==>
2*pow2 2 ==>
2*(2*pow2 1) ==>
2*(2*(2*pow2 0)) ==>
2*(2*(2*1)) ==> 8
```

Recursion 1: integer

e.g. find `x to the power n`

```
base case: n=0 ==> 1

recursion case: n>0 ==> mutiply x by xn-1

- fun power _ 0 = 1 |
      power x n = x*power x (n-1);

> val power = fn : int -> int -> int

1 is int so x*power x (n-1) must be int so *
must be int multiplication so x must be int

0 is int so n must be int
```

Recursion 1: integer

```
- power 3 3;
> 27 : in
power 3 3 ==>
3*power 3 2 ==>
3*(3*power 3 1) ==>
3*(3*(3*power 3 0)) ==>
3*(3*(3*1)) ==> 27
```

Recursion 1

e.g. make a string with `n "."s`

base case: n=0 ==> ""

recursion case: n>0 ==> join "." to n-1 "."s

```
- fun stops 0 = "" |
      stops n = "."^stops (n-1);
> val stops = fn : int -> string
```

0 is int so n must be int

"" is string so "."^stops (n-1) must be string so ^ must return string (!) so "." must be string (!) so stops (n-1) must be string

Recursion 1: integer

```
- stops 3;
> "..." : string
stops 3 ==>
"."^stops 2 ==>
"."^("."^stops 1) ==>
"."^("."^("."^stops 0)) ==>
"."^("."^("."^"")) ==>
"..."
```

Recursion 1: integer

e.g. make a string with `n lots of string s`

`base case: n=0 ==> “”`

`recursion case: n>0 ==> join s to n-1 s’s`

```
- fun sJoin _ 0 = “” |
    sJoin s n = s^sJoin s (n-1);
> val sJoin =
   fn : string -> int -> string
```

Recursion 1: integer

```
- sJoin 3 “run”;
> “runrunrun” : int
sJoin 3 “run” ==>
“run”^sJoin 2 “run” ==>
“run”^(“run”^sJoin 1 “run”) ==>
“run”^(“run”^(“run”^sJoin 0 “run”)) ==>
“run”^(“run”^(“run”^””)) ==>
“runrunrun”
```

Partial application 1

apply a function of n formal parameters to m<n-1 actual parameters

forms a new function of n-m formal parameters

like original function

with formal parameters 1..m *frozen* to actual parameters 1..m

Partial application 1

```
- fun f a b c d = ...
> val f =
    fn : type1 -> type2 -> ... typeN
- val aval = ...;
> val aval = ...: type1
- val f' = f aval;
> val f' : type2 -> ... typeN
```

f' is like f with a frozen to the value of aval

Partial application 1

e.g. powers

```
- fun power _ 0 = 1 |
      power x n = x*power x (n-1);
> val power = fn : int -> int  -> int
- val pow2 = power 2;
> val pow2 = fn : int -> int
pow2 is like power with x frozen to 2
- pow2 3;
> 8 : int
pow2 3 ==> power 2 3 ==> ... 8
```

Partial application 1

e.g. `sJoin`

```
- fun sJoin _ 0 = "" |
      sJoin s n = s^sJoin s (n-1);
> val sJoin = fn : string -> int -> string
- val stops = sJoin ".";
> val stops = fn : int -> string
stops is like sJoin with s frozen to "."
- stops 3;
> "..." : string
stops 3 ==> sJoin "." 3 ==> ... "..."
```

Conditional expression

pattern matching with constant can detect presence or absence of value

can't check properties of value

conditional expression

`if` *`expression1`*

`then` *`expression2`*

`else` *`expression3`*

`expression1` `must return bool`

`expression2` `and` *`expression3`* `must return same type`

Conditional expression

e.g. find absolute value of integer

```
- fun abs x =
        if x>=0
        then x
        else ~x;
> val abs = fn : int -> int
```

Conditional expression

e.g. find bigger of two integers

```
- fun max (x:int) y =
        if x>y
        then x
        else y;
> val max = fn : int -> int -> int
```

NB > is overloaded so must make type of one of x or y explicit