SML Functions

SML functions are values.

1. defining functions
2. multiple argument functions
3. higher-order functions
4. currying
5. polymorphic functions

Defining Functions

Defining integer values, etc:

```
- val i = 3;
val i = 3 : int

- val s = "abc";
val s = "abc" : string
```

Defining function values:

```
- val inc = fn (x) => x + 1;
val inc = fn : int -> int

- inc(3);
val it = 4 : int

- val is_it_3 = fn (x) => if x = 3 then "yes" else "no";
val is_it_3 = fn : int -> string

- is_it_3(4);
val it = "no" : string
```

Fun with fun

The previous definitions can be abbreviated:

`fun` <identifier>(<parameter list>) `=` <expression>`;`

Examples:

```
- fun inc(x) = x + 1;
val inc = fn : int -> int

- fun is_it_3(x) = if x = 3 then "yes" else "no";
val is_it_3 = fn : int -> string

- fun test(x,y) = if x < y then y+1 else x+1;
val test = fn : int * int -> int
```

Functions as Values

Function types:

`fn:` <domain type> `->` <range type>

Examples:

```
- 3;
val it = 3 : int

- fn (x,y) => x + y;              (* anonymous function *)
val it = fn : int * int -> int

- val p = (fn (x,y) => x + y, fn (x,y) => x - y);
val p = (fn,fn) : (int * int -> int) * (int * int -> int)

- #1(p)(2,3);
val it = 5 : int

- #2(p)(2,3);
val it = ~1 : int
```

Functions can be tuple components.

Functions as Values

More examples:

```
- fun add1(x) = x + 1;
val add1 = fn : int -> int
- fun add2(x) = x + 2;
val add2 = fn : int -> int
- fun add3(x) = x + 3;
val add3 = fn : int -> int

- val l = [add1,add2,add3];
val l = [fn,fn,fn] : (int -> int) list

- hd(l)(3);
val it = 4 : int
- hd(tl(l))(3);
val it = 5 : int
```

Functions can be list elements.

Functions as Values

More examples: higher-order functions

```
- fun do_fun(f,x) = f(x) + x + 1;
val do_fun = fn : (int -> int) * int -> int

- do_fun(add2,3);
val it = 9 : int

- do_fun(add3,5);
val it = 14 : int

- fun make_addx(x) = fn(y) => y + x;
val make_addx = fn : int -> int -> int

- val add5 = make_addx(5);
val add5 = fn : int -> int

- add5(3);
val it = 8 : int
```

A higher-order function:

- "processes" other functions
- takes a function as input, or returns a function as a result

Functions as Values

In SML, functions are *first-class* citizens.

Just like any other value: they can be

- placed in tuples
- placed in lists
- passed as function arguments
- returned as function results

Compare with C

We must use function pointers (and it's ugly):

```
#include <stdio.h>

int add3(int x)
{
  return x + 3;
}

int do_fun(int (*fp)(int x), int y)
{
  return (*fp)(y) + y + 1;
}

void main(void)
{
  printf("%d\n",do_fun(add3,5));
}
```

Compare with Pascal

A little better, but we can't return functions as a result.

```
function add3(x : integer): integer;

begin
  add3 := x + 3;
end;

function do_fun( f (x : integer): integer;
                 y: integer): integer;

begin
  do_fun := f(y) + y + 1;
end;

begin
  writeln(do_fun(add3,5));
end.
```

Multiple Argument Functions

- In reality, each SML function takes exactly one argument and returns one result value.
- If we need to pass multiple arguments, we generally package the arguments up in a tuple.

```
- fun add3(x,y,z) = x + y + z;
val add3 = fn : int * int * int -> int
```

- If a function takes n argument, we say that it has *arity* n.

Multiple Argument Functions

Can we implement "multiple argument functions" without tuples or lists?

- Yes, use higher-order functions

```
- fun add3(x) = fn (y) => fn (z) => x + y + z;
val add3 = fn : int -> int -> int -> int

- ((add3(1))(2))(3);
val it = 6 : int

- add3 1 2 3;           (* omit needless parens *)
val it = 6 : int

(* abbreviate definition *)

- fun add3 x y z = x + y + z;
val add3 = fn : int -> int -> int -> int

- add3 1 2 3;
val it = 6 : int
```

Multiple Argument Functions

Look closely at types:

1. `fn : int -> int -> int -> int`

abbreviates

2. `fn : int -> (int -> (int -> int))`

which is different than

3. `fn : (int -> int) -> (int -> int)`

- The first two types describes a function that
  - takes an integer as an argument and returns a function of type `int -> int -> int` as a result.
- The last type describes a function that
  - takes a function of type `int -> int` as an argument and returns a function of type `int -> int` as a result.

Currying

The function

```
- fun add3(x) = fn (y) => fn (z) => x + y + z;
val add3 = fn : int -> int -> int -> int
```

is called the "curried" version of

```
- fun add3(x,y,z) = x + y + z;
val add3 = fn : int * int * int -> int
```

History:

- The process of moving from the first version to the second is called "currying" after the logician Haskell Curry who supposedly first identified the technique.
- Some people say that another logician named Skolefinkel actually invented it, but we still call it "currying" (thank goodness!).

Curried Functions

Curried functions are useful because they allow us to create “partially instantiated” or “specialized” functions where some (but not all) arguments are supplied.

Example:

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

- val add3 = add 3;
val add3 = fn : int -> int

- val add5 = add 5;
val add5 = fn : int -> int

- add3 1 + add5 1;
val it = 10 : int

- add3(1) + add5(1);
val it = 10 : int
```

The last example shows that parens around arguments are not always needed (having them is just our convention).

Polymorphic Functions

The theory of polymorphism underlying SML is an elegant feature that clearly distinguishes SML from other languages that are less well-designed.

Example:

```
- fun id x = x;
val id = fn : 'a -> 'a
- id 5;
val it = 5 : int
- id "abc";
val it = "abc" : string
- id (fn x => x + x);
val it = fn : int -> int
- id(2) + floor(id(3.5));
val it = 5 : int
```

Polymorphism: (poly = many, morph = form)

Polymorphic Functions

More examples:

```
- hd;
val it = fn : 'a list -> 'a

- hd [1,2,3];
val it = 1 : int

- hd ["a","b","c"];
val it = "a" : string

- val hd_int = hd : int list -> int;
val hd_int = fn : int list -> int

- hd_int [1,2,3];
val it = 1 : int
```

Polymorphic Functions cont.

More examples:

```
- val hd_int = hd : int list -> int;
val hd_int = fn : int list -> int

- hd_int ["a","b","c"];
stdIn:29.1-29.21 Error: operator and operand don't...

- length;
val it = fn : 'a list -> int

- (id,id);
val it = <poly-record> : ('a -> 'a) * ('b -> 'b)
```

Polymorphism

- Think of `fn : 'a -> 'a` as the type of a function that has many different versions (one for each type).
- `'a` is a *type variable* — a place holder where we can fill in any type.
- We can have more than one type variable in a given type

```
- val two_ids = (id,id);
val two_ids = <poly-record> : ('a -> 'a) * ('b -> 'b)

- val two_ids = (id : int -> int, id : char -> char);
val two_ids = (fn,fn) : (int -> int) * (char -> char)

- val two_ids = (id : int -> char, id : char -> char);
stdIn:15.4-31.11 Error: expression doesn't match ....
  expression: int -> int
  constraint: int -> char
  in expression:
    id: int -> char
```

- Note that the SML implementation always comes up with the most general type possible (but we can override with a specific type declaration).
- A type with no type variables is also called a *ground type*.
- There are many subtle and interesting points about polymorphism that we will come back to later.

A Higher-order Polymorphic Function

Compose: `o` (pre-defined function)

```
- val add8 = add3 o add5;
val add8 = fn : int -> int
- add8 3;
val it = 11 : int
- (op o);
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Note `(op o)` forces an infix operator to act like regular non-fix function identifier.

User-defined version:

```
- fun my_o (f,g) = fn x => f(g(x));
val my_o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Terms and Concepts

currying	polymorphism	higher-order function
arity	compose	type variable
function application	partially instantiated	

- define higher-order function and give an example
- define polymorphic function and give an example
- be able to define functions using both `fun` and `fn`
- given an uncurried function, define the curried version (and vice versa).