Defining Functions

Defining values of simple types

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

Defining function values:

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

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

- val is_3 = fn x =>
      if x = 3 then "yes" else "no";
val is_3 = fn : int -> string

- is_3 4;
val it = "no" : string
```

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

Fun with fun

Amtoft
from Hatcliff
from Leavens

The previous definitions can be abbreviated:

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

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

- fun is_3 x =
    if x = 3 then "yes" else "no";
val is_3 = fn : int -> string

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

ML Programs

A (simple) ML program is generally a sequence of function definitions

```
fun push (value, stack)
    ...
    ...;

fun pop (stack)
    ...
    ...;

fun empty (stack)
    ...
    ...;

fun make-stack (value)
    ...
    ...;
```

Functions as Values

Functions can be anonymous

```
- fn x => x + 2;
val it = fn : int -> int
```

Functions can be tuple components

```
- 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 as Values

Functions can be list elements

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

- hd(ls)(3);
val it = 4 : int

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

Higher-Order Functions

Functions can be given as arguments

```
- 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
```

Functions can be returned as results

```
- 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
```

Functions Are Values

A higher-order function

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

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

Amtoft
from Hatcliff
from Leavens

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.
```

Scope of Variables

```
- val a = 2;
val a = 2 : int
- fun myfun x = x + a;
val myfun = fn : int -> int
- val a = 4;
val a = 4 : int
- myfun(5);
???
```

```
val it = 7 : int
```

- Declarations at the top-level may seem like assignments.... but they're not!
- Technically speaking, ML is statically scoped
- New definitions of the same variable don't overwrite old definitions; they *shadow* the old definitions
- For efficiency, old definitions may be garbage collected if they are not referred to.

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
```

Interpreting Function Types

Look closely at types:

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

abbreviates

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

which is different from

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 argument and returns a function of type int -> int.

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.
- The technique actually goes back to another logician named Schönfinkel
- but we still call it "currying" (thank goodness!).

Instantiating 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.

```
- 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 2 + add5 6;
val it = 16 : int
```

Polymorphic Functions

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

```
- 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 and Monomorphic Functions


```
- 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

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

Polymorphism


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

- val two_id = (id : int -> int, id)
val two_id = (fn,fn) : (int -> int) * ('a -> 'a)
```

- 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.
- A type can contain more than one type variable
- 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 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); (* convert infix to non-infix *)
val it = fn :
   ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

User-defined version:

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