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";


- is_it_3(4);
```

Fun with fun

The previous definitions can be abbreviated:

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

Examples:

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


- fun is_it_3(x) = if x = 3 then "yes" else "no";


- fun test(x,y) = if x < y then y+1 else x+1;
```

Function types:

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

Functions as Values

Examples:

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

- fn (x,y) => x + y;                     (* anonymous function *)


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


- #1(p)(2,3);


- #2(p)(2,3);
```

Functions can be tuple components.

Functions as Values

More examples:

```
- fun add1(x) = x + 1;
val add2 = 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];


- hd(l)(3);

- hd(tl(l))(3);
```

Functions can be list elements.

Functions as Values

More examples: higher-order functions

```
- fun do_fun(f,x) = f(x) + x + 1;
```

```
- do_fun(add2,3);
```

```
- do_fun(add3,5);
```

```
- fun make_addx(x) = fn(y) => y + x;
```

```
- val add5 = make_addx(5);
```

```
- add5(3);
```

A higher-order function:

- ...
- ...

Functions as Values

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

Just like any other value: they can be

- ...

- ...

- ...

- ...

Compare with C

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

```c
#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 a function takes n argument, we say that it has ...

Multiple Argument Functions

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

- ...

  –

  - ((add3(1))(2))(3);

  - add3 1 2 3;   (* omit needless parens *)

  (* 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
  - ...

- The last type describes a function that
  - ...

Currying

The function

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

is called the .... version of

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

History:

- The process of moving from the second version to the first is called "currying" after the logician Haskell Curry who supposedly first identified the technique.

- Some people say that another logician named Schönfinkel 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 add5 = add 5;


- add3 1 + add5 1;


- add3(1) + add5(1);
```

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;

- id 5;

- id "abc";

- id (fn x => x + x);

- id(2) + floor(id(3.5));
```

Polymorphism: (poly = many, morph = form)

Polymorphic Functions

More examples:

```
- hd;


- hd [1,2,3];


- hd ["a","b","c"];


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


- hd_int [1,2,3];


- hd_int ["a","b","c"];


- length;


- (id,id);
```

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;
```

```
- add8 3;
```

```
- (op o);
```

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

User-defined version:

```
-
```

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).