Functional Programming & SML

Functional Programming

- Based on mathematical functions
  - Expression, recursive functions, non-updateable variables
  - Side effect-free programming
- Mathematical functions vs. Functions in IPL?
  - No side effects vs side effects

Referential Transparency

- In FPs, the evaluation of a function always produces the same result given the same parameters.
  - **Syntactically the same, then the same semantics (meaning)**
- Why?
- Why useful?
  - Equational reasoning

Characteristics of Functional Languages

- First-class function values
- Higher-order functions
- Nonstrict functions and Lazy evaluation
- Parametric polymorphism
- Garbage collection

Functions as First Class Values

- In FLs, functions are **first-class** values.
  - Functions can be passed to a function as actual parameters.
  - Functions can be returned from a function call as results.
  - Functions can be stored into composite values (data structures).
  - Functions can be created dynamically.

Functions as First Class Values

- In IPLs, functions are **second-class** values!
  - Restrictions on use

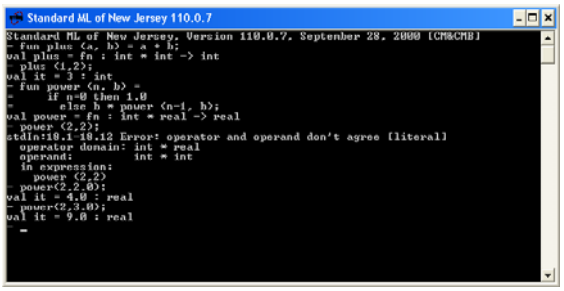
Creating Functions

- How to **conveniently** define (specify) functions?
  - **Curried function** and **Partial application**

Normal Functions

```
fun plus (a, b) = a + b;
plus: (int, int) → int
plus(1, 2) => 3

fun power (n, b) =
    if n=0 then 1.0
         else b * power (n-1, b)
power : int * real →real
power (2, x) => x²
power (3, x) => x³
```

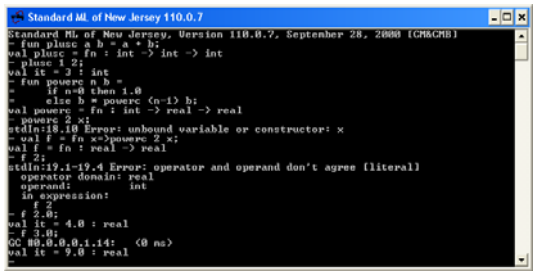

Normal Functions in SML



Curried Functions

```
fun plusc a b = a + b;
plusc: int → int → int
Plusc 1 2  => 3

fun powerc n b =
    if n=0 then 1.0
    else b * powerc (n-1) b
powerc : (int, real) → real
powerc 2 x => x²
powerc 3 x => x³
```


Curried Functions in SML



Partial Application

```
fun plusc a b = a + b;
plus: int → int → int
plusc 1 2  => 3

plusc 1 => function (f b = 1 + b)

plusc 2 => function (g b = 2 + b)

…
```

Partial Application in SML

```
Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM&CMB]
- fun plusc a b = a + b;
val plusc = fn : int -> int -> int
- val add1 = plusc 1;
val add1 = fn : int -> int
- add1 100;
val it = 101 : int
- val add2 = plusc 2;
val add2 = fn : int -> int
- add2 100;
val it = 102 : int
-
```


Partial Application

```
fun powerc n b =
    if n=0 then 1.0
    else b * powerc (n-1) b

powerc : int → real → real

powerc 2 => function (f b = powerc 2 b)

powerc 3 => function (g b = powerc 3 b)

…
```


Partial Application in SML

```
Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM&CMB]
- fun powerc n b =
=     if n=0 then 1.0
=     else b * powerc (n-1) b
= ;
val powerc = fn : int -> real -> real
- val sqr = powerc 2;
val sqr = fn : real -> real
- sqr 100.0;
val it = 10000.0 : real
- val cube = powerc 3;
val cube = fn : real -> real
- cube 100.0;
val it = 1000000.0 : real
-
```


Mathematical Functions - Theory

- **Lambda calculus**
  - Lambda expressions describe nameless functions.
  - Can be used for creating function dynamically.

```
lambda (a, b) (a + b)
```


Types of Functions

- First-order function
  - A function whose parameters/result are non-functional values.
- Second-order function
  - A function whose parameters/result can be first-order functions.
- Third-order function
  - A function whose parameters/result can be second-order functions.
- …

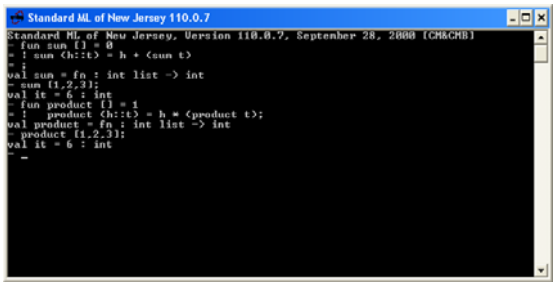
Higher-Order Functions

- **High-order function**
  - Any function higher than first-order function.
  - A function that takes a function as an argument or returns a function as a result.
- Why useful?
  - Higher degree of abstraction!
  - Software reuse

FOF Example

```
sum :: [int] → int
fun sum [] = 0
| sum (h::t) = h + (sum t)

product :: [int] → int
fun product [] = 1
|   product (h::t) = h * (product t)
```


FOF Example in SML

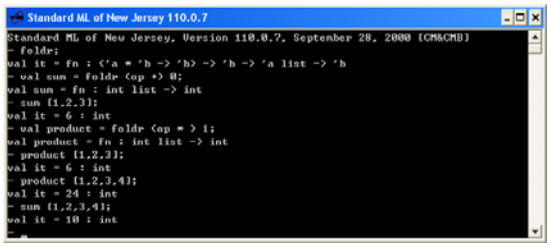


HOF Example

```
foldr :: (int→int→int)→int→[int]→int
fun foldr op n = f
            where
            f [] = n
            f (h::t) = h op (f t)
```

```
sum = foldr (+) 0
product = foldr (*) 1
…
```


HOF Example in SML



Semantics of Functions

- Strict functions
- Nonstrict functions

Strict Functions

- A function is said to be **strict** if a call to this function can be evaluated only if all of its arguments can be evaluated.
  - It requires all of its arguments to be defined.

Strict Function Example

```
g x a b c = if (x<0) then a
            elseif (x=0) then b
            else c

(g exp1 exp2 exp3 exp4)

/* Assume exp2 and exp4 do not terminate */
(g 0 exp2 100 exp4)↑
```


Nonstrict Functions

- A function is said to be **non-strict** if it does not impose the requirement:
  - A call to this function can be evaluated even when not all of the arguments can be evaluated.
  - It does not require all of its arguments to be defined.

Nonstrict Function Example

```
g x a b c = if (x<0) then a
            elseif (x=0) then b
            else c

(g exp1 exp2 exp3 exp4)

/* Assume exp2 and exp4 do not terminate */
(g 0 exp2 100 exp4) => 100 ↓
```


Evaluation Order

- Given a function application (call), whether or not to evaluate actual parameters before passing them to the function?
  - Evaluate before passed
    - **Applicative order evaluation**
  - Passed unevaluated & evaluate when it is needed
    - **Normal order evaluation**

Evaluation Order

- Applicative order evaluation
  - Implement **strict** functions
- Normal order evaluation
  - Implement **non-strict** functions

Applicative Order Evaluation

```
g x a b c = if (x<0) then a
            elseif (x=0) then b
            else c

/* Assume exp2 and exp4 do not terminate */
(g 0 exp2 100 exp4)↑

(g –1 (1+2) (2+3) (3+4)) => 3
```

Normal Order Evaluation

```
g x a b c = if (x<0) then a
            elseif (x=0) then b
            else c

/* Assume exp2 and exp4 do not terminate */
(g 0 exp2 100 exp4) => 100 ↓

(g –1 (1+2) (2+3) (3+4)) => 3
```


Lazy Evaluation

- Normal order evaluation
  - Evaluate it when it is needed!
- Lazy evaluation = Normal-order evaluation+ Evaluate at first use & store for later uses. (Memoized)
  - Optimization of the normal-order evaluation method.
  - Evaluate it when it is needed, but only once!

Lazy Evaluation Example

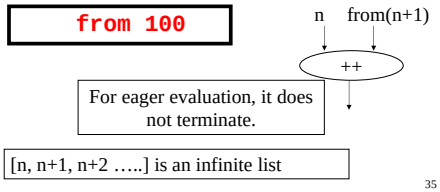
```
f x = x + x

/* Normal-order evaluation */
f (1+2) => 3

/* Lazy evaluation */
f (1+2) => 3
```


Infinite (Lazy) Data Structures

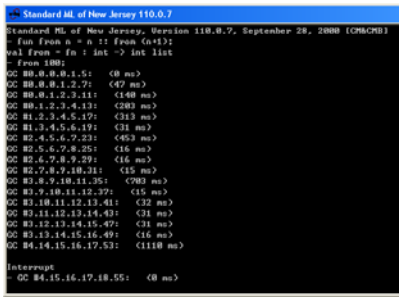
- We can build **infinite (lazy) data structures** using LE.

Infinite (Lazy) Lists

```
fun from n = n ++ from (n+1)
where
++ 1 [2,4] = [1,2,4]
```



Strict Lists in sml



Using Infinite (Lazy) Lists

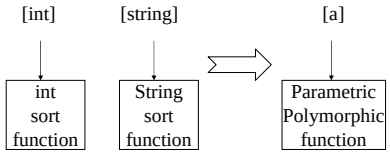
```
fun from n = n ++ from (n+1)
where
++ 1 [2,4] = [1,2,4]

fun first_prime ( n: ns) =
    if prime (n) then n
    else first_prime ns

first_prime (from 10)
first_prime (from 1000)
```


Lazy Evaluation with IL?

- Question: **lazy evaluation +Imperative Language?**
  - Yes, but … Side effects!
- So,
  - **Pure FL**
    - **Lazy evaluation (Miranda, Haskell)**
  - **Hybrid FL**
    - **Eager evaluation (ML, LISP, Scheme)**

Parametric Polymorphism

- Function that operates uniformly on arguments of a whole family of related types.
  - Not a single type!



Parametric Polymorphic Functions

```
fun snd1 (x :int, y : int)
    return (y)
```

snd1 (1, 10)
? snd1(1, true)

```
fun snd2 (x: int, y : bool)
    return (y);
```

snd2 (1, true)
? snd1('a', true)

```
fun snd3 (x: char, y: bool)
    return (y);
```

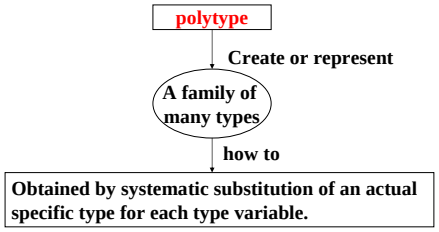
snd3 ('a', true)
? snd3 ("A", true)

```
fun snd (x: α, y: β)
    return (y)
```


Polytype

- Parametric polymorphic function
  - Type is not fixed! (Type variables)
  - **`snd: α X β -> β`**
- **Polytype** is a type that contain one or more type variable.
  - **`int X int -> int`** ==> Monotype
  - **`α X β -> β`** ==>Polytype
- Polymorphic typing vs. Monomorphic typing

Polytype



Polytype Example

```
snd: α X β -> β
```

int x int -> int
int x real -> real
int x bool -> bool
…….

int x real -> int ?
int x int -> real ?

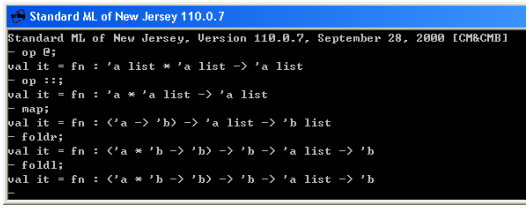
Polytype Example

```
concat :: [[int]] -> [int]
concat [] = []
concat (h:t) = h ++ (concat t)

all :: [bool] -> bool
all [] = True
all (h:t) = h && (all t)
```


Polytype Example

```
foldr :: (int→int→int)→int→[int]→int
foldr op n = f
             where
             f [] = n
             f (h:t) = h `op` (f t)

foldr :: (a -> b -> b) -> b -> [a] -> b

concat = foldr (++) []
all = foldr (&&) True
```


Polytype Example



Polymorphism in PLs !

Types of Polymorphisms

- Ad hoc polymorphism
  - Overloading
  - Coercion
- Universal polymorphism
  - Parametric polymorphism
  - Inclusion polymorphism
- Generics

Ad Hoc Polymorphism

- **Overloading**
  - The ability of a single identifier or operator to denote several abstractions (functions/procedures) simultaneously.
  - The compiler resolves!
- **Coercion**
  - The ability of omitting semantically necessary type conversions.
  - The programmer should know about allowed coercion rules!

Universal Polymorphism

- **Parametric polymorphism**
  - The ability of abstractions that operate uniformly on values of different types.
- **Inclusion polymorphism**
  - The ability of subtypes to inherit operations from their supertypes.

Generics

- **Generics** (templates) is an abstraction over declarations.
  - The actual declaration will be created by a compiler via instantiation.
- Parametric polymorphism vs Generics?

SML: A Functional Language

Pattern-Matching Example

```
fun fact n =
  if n = 0 then 1 else n * fact(n-1);
```

```
fun fact 0 = 1
|   fact n = n * fact(n-1);
```

```
fun reverse L =
   if null L then nil
   else reverse(tl L) @ [hd L];
```

```
fun reverse nil = nil
|   reverse (first::rest) =
       reverse rest @ [first];
```


Pattern-Matching Example

```
fun f nil = 0
|   f (first::rest) = first + f rest;
```

```
fun f nil = 0
|   f (true::rest) = 1 + f rest
|   f (false::rest) = f rest;
```

```
fun f nil = nil
|   f (first::rest) = first+1 :: f rest;
```

Function Values

- Named function:

```
- fun f x = x + 2;
val f = fn : int -> int
- f 1;
val it = 3 : int
```

- Anonymous function:

```
- fn x => x + 2;
val it = fn : int -> int
- (fn x => x + 2) 1;
val it = 3 : int
```


Function Values

- We can define what **`fun`** does in terms of **`val`** and **`fn`**
- These two definitions have the same effect:
  - **`fun f x = x + 2`**
  - **`val f = fn x => x + 2`**

Curried Functions

```
- fun f (a,b) = a+b;
val f = fn : int * int -> int
- fun g a = fn b => a+b;
val g = fn : int -> int -> int
- f(2,3);
val it = 5 : int
- g 2 3;
val it = 5 : int
```


Curried Functions

```
- val add2 = g 2;
val add2 = fn : int -> int
- add2 3;
val it = 5 : int
- add2 10;
val it = 12 : int
```

```
- quicksort (op <) [1,4,3,2,5];
val it = [1,2,3,4,5] : int list
- val sortBackward = quicksort (op >);
val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
val it = [5,4,3,2,1] : int list
```


Multiple Curried Parameters

```
- fun f (a,b,c) = a+b+c;
val f = fn : int * int * int -> int
- fun g a = fn b => fn c => a+b+c;
val g = fn : int -> int -> int -> int
- f (1,2,3);
val it = 6 : int
- g 1 2 3;
val it = 6 : int
```


Easier Notation for Currying

- Instead of writing:

```
fun f a = fn b => a+b;
```

- We can just write:

```
fun f a b = a+b;
```

- This generalizes for any number of curried arguments

```
- fun f a b c d = a+b+c+d;
val f = fn : int -> int -> int -> int -> int
```

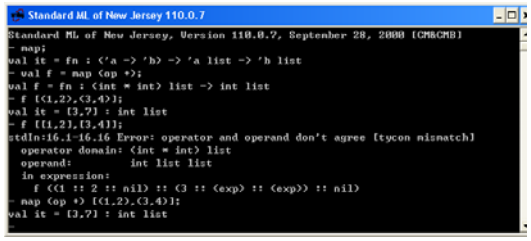
The `map` Function

```
- map ~ [1,2,3,4];
val it = [~1,~2,~3,~4] : int list
- map (fn x => x+1) [1,2,3,4];
val it = [2,3,4,5] : int list
- map (fn x => x mod 2 = 0) [1,2,3,4];
val it = [false,true,false,true] : bool list
- map (op +) [(1,2),(3,4),(5,6)];
val it = [3,7,11] : int list
```


The `map` Function Is Curried

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- val f = map (op +);
val f = fn : (int * int) list -> int list
- f [(1,2),(3,4)];
val it = [3,7] : int list
```


The `map` Function



The `foldr` Function

- So **`foldr (op +) 0 [1,2,3,4]`** evaluates as 1+(2+(3+(4+0)))=10

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
- foldr (op * ) 1 [1,2,3,4];
val it = 24 : int
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldr (op ::) [5] [1,2,3,4];
val it = [1,2,3,4,5] : int list
```


The `foldr` Function Is Curried

```
- foldr;
val it =
fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr (op +);
val it = fn : int -> int list -> int
- foldr (op +) 0;
val it = fn : int list -> int
- val addup = foldr (op +) 0;
val addup = fn : int list -> int
- addup [1,2,3,4,5];
val it = 15 : int
```


The `foldl` Function

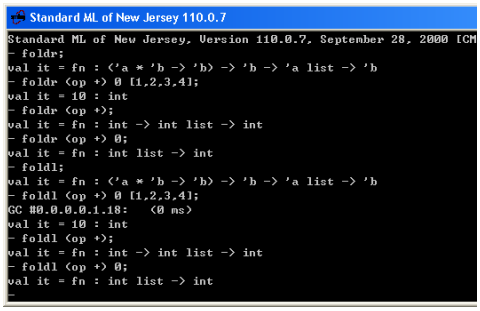
- So **`foldl (op +) 0 [1,2,3,4]`** evaluates as 4+(3+(2+(1+0)))=10

```
- foldl (op +) 0 [1,2,3,4];
val it = 10 : int
- foldl (op * ) 1 [1,2,3,4];
val it = 24 : int
```

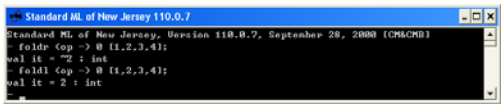
foldr Vs foldl

```
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldl (op ^) "" ["abc","def","ghi"];
val it = "ghidefabc" : string
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```


foldr Vs foldl



foldr Vs foldl



User-defined Data Types

```
-datatype Direction = N | E | S | W;
datatype Direction = N | E | S | W

-fun heading N = 0.0 |
=       heading E = 90.0 |
=       heading S = 180.0 |
=       heading W = 270.0 ;
val heading = fn : Direction -> real

-datatype 'a BST = Nil |
=           Node of 'a * 'a BST * 'a BST;
datatype 'a BST = Nil |
          Node of 'a * 'a BST * 'a BST
```