

Functional Programming & SML

CS216

1

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

CS216

2

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

CS216

3

Characteristics of Functional Languages

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

CS216

4

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.

CS216

5

Functions as First Class Values

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

CS216

6

Creating Functions

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

CS216

7

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
```

CS216

8

Normal Functions 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 plus a, b = a + b;
- val plusc fn : int = int -> int
- val plusc <1,2>;
- val it = 3 : int
- fun power n, b =
  if n=0 then 1.0
  else b * powerc (n-1, b);
- val powerc fn : int * real -> real
- powerc (2,2)
- val it = 4.0 : real
- powerc (2,3.0);
- val it = 9.0 : real
-
```

CS216

9

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
```

CS216

10

Curried Functions 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 plusc <1,2>;
- val it = 3 : int
- fun powerc n b =
  if n=0 then 1.0
  else b * powerc (n-1) b;
- val powerc fn : int -> real -> real
-
stdIn:18.10 Error: unbound variable or constructor:
- val f = fn : x=>powerc 2 x;
val f = fn : real -> real
- f 2;
stdIn:19.1-19.4 Error: operator and operand don't agree (literal)
  operand:   int
  in expression:
  operator:  int
- f 2.0;
val it = 4.0 : real
QC #0.0.0,0.1.14: <0 ns>
val it = 9.0 : real
-
```

CS216

11

Partial Application

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

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

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

...
```

CS216

12

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
-
```

CS216

13

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)
...

```

CS216

14

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
-
```

CS216

15

Mathematical Functions - Theory

- **Lambda calculus**

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

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

CS216

16

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.
- ...

CS216

17

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

CS216

18

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)
```

CS216

19

FOF Example in SML

```
Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 (CH&CRB)
- fun sum [] = 0
- | sum (h::t) = h + (sum t)
val sum = fn : int list -> int
- sum [1,2,3];
val sum = 6 : int
- fun product [] = 1
- | product (h::t) = h * (product t);
val product = fn : int list -> int
product [1,2,3];
val product = 6 : int
-
```

CS216

20

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
...
```

CS216

21

HOF Example in SML

```
Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 (CH&CRB)
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- val sum = foldr (op +) 0;
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int
- val product = foldr (op *) 1;
val product = fn : int list -> int
product [1,2,3];
val it = 6 : int
- product [1,2,3,4];
val it = 24 : int
- sum [1,2,3,4];
val it = 10 : int
-
```

CS216

22

Semantics of Functions

- Strict functions
- Nonstrict functions

CS216

23

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.

CS216

24

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)↑
```

CS216

25

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.

CS216

26

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 ↓
```

CS216

27

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

CS216

28

Evaluation Order

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

CS216

29

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
```

CS216

30

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

```

CS216

31

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!

CS216

32

Lazy Evaluation Example

```

f x = x + x

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

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

```

CS216

33

Infinite (Lazy) Data Structures

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

CS216

34

Infinite (Lazy) Lists

```

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

```

from 100

For eager evaluation, it does
not terminate.

CS216

35

Strict Lists in sml

```

-- Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 (CM/CMU)
- fun from n = n :: from (n+1);
val From = fn : int => int list
  val _ =
    let
      fun loop 0 = []
        | loop n = (fn () => print (Int.toString n ^ " ")) :: loop (n+1)
    in
      loop 0
    end
  #> B0.0.0.0.1.5: <(0 ms)>
  #> B0.0.0.1.2.7: <(47 ms)>
  #> B0.0.1.2.3.11: <(1 ms)>
  #> B0.0.1.2.3.11: <(1 ms)>
  #> B0.1.2.3.4.5.19: <(13 ms)>
  #> B1.1.4.5.6.19: <(31 ms)>
  #> B2.4.5.6.23: <(453 ms)>
  #> B3.6.7.8.9.29: <(51 ms)>
  #> B2.7.8.9.10.31: <(16 ms)>
  #> B3.8.9.10.11.35: <(793 ms)>
  #> B3.8.9.10.11.35: <(16 ms)>
  #> B1.10.11.12.13.41: <(21 ms)>
  #> B0.11.12.13.14.43: <(31 ms)>
  #> B0.12.13.14.15.47: <(31 ms)>
  #> B0.13.14.15.16.49: <(16 ms)>
  #> B0.14.15.16.17.53: <(1118 ms)>

```

CS216

36

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)

```

CS216

37

Lazy Evaluation with IL?

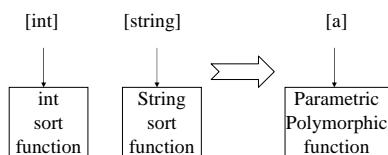
- Question: **lazy evaluation +Imperative Language?**
 - Yes, but ... Side effects!
- So,
 - **Pure FL**
 - Lazy evaluation (Miranda, Haskell)
 - **Hybrid FL**
 - Eager evaluation (ML, LISP, Scheme)

CS216

38

Parametric Polymorphism

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



CS216

39

Parametric Polymorphic Functions

```

fun snd1 (x : int, y : int)      snd1 (1, 10)
    return (y)                  ? snd1(1,true)

fun snd2 (x: int, y : bool)      snd2 (1, true)
    return (y);                 ? snd1('a',true)

fun snd3 (x: char, y: bool)      snd3 ('a', true)
    return (y);                 ? snd3 ("A",true)

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

```

CS216

40

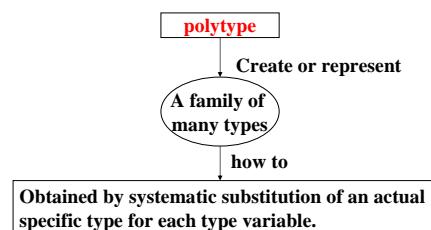
Polytype

- Parametric polymorphic function
 - Type is not fixed! (Type variables)
 - **snd**: $\alpha \times \beta \rightarrow \beta$
- **Polytype** is a type that contain one or more type variable.
 - $\text{int} \times \text{int} \rightarrow \text{int}$ ==> Monotype
 - $\alpha \times \beta \rightarrow \beta$ ==> Polytype
- Polymorphic typing vs. Monomorphic typing

CS216

41

Polytype



CS216

42

Polytype Example

```
snd ::  $\alpha$  x  $\beta \rightarrow \beta$ 
```

```
int x int -> int
int x real -> real
int x bool -> bool
.....
```

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

CS216

43

Polytype Example

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

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

CS216

44

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
```

CS216

45

Polytype Example

```
 Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM&CMB]
- op @;
val it = fn : 'a list * 'a list -> 'a list
- op ::;
val it = fn : 'a * 'a list -> 'a list
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

CS216

46

Polymorphism in PLs !

CS216

47

Types of Polymorphisms

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

CS216

48

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!

CS216

49

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.

CS216

50

Generics

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

CS216

51

SML: A Functional Language

CS216

52

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];
```

CS216

53

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;
```

CS216

54

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
```

CS216

55

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
```

CS216

56

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
```

CS216

57

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
```

CS216

58

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
```

CS216

59

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
```

CS216

60

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
```

CS216

61

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
```

CS216

62

The map Function

A screenshot of a terminal window titled "Standard ML of New Jersey 110.0.7, Version 110.0.7, September 28, 2000 (GM6CM8)". The window shows the following code and an error message:

```
- map;
val it = fn : <'a -> 'b> -> 'a list -> 'b list
- val F = map (op +);
val F = fn : <int * int> list -> int list
- map [(1,2),(3,4)];
val it = [3,7] : int list
f [(1,2),(3,4)];
stdIn[16,1]=16.16 Error: operator and operand don't agree [tycon mismatch]
operator domain: <int * int> list
operands:
  int list list
  in expression:
    f <(1 :: 2 :: nil) :: (3 :: (exp :: (exp :: (exp :: nil
map (op +) [(1,2),(3,4)];
val it = [3,7] : int list
"
```

CS216

63

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 = "abcdefghijklm" : string
- foldr (op ::) [5] [1,2,3,4];
val it = [1,2,3,4,5] : int list
```

CS216

64

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
```

CS216

65

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
```

CS216

66

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
```

CS216

67

foldr Vs foldl

```
# Standard ML of New Jersey 110.0.7, September 28, 2000 (CM)
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr (op =>) @ [1,2,3,4];
val it = 1@ : int
- foldr (op =>);
val it = fn : int -> int list -> int
- foldr (op => @);
val it = fn : int list -> int
- foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldl (op =>) @ [1,2,3,4];
GC #0,0,0,1,18: <0 ms>
val it = 1@ : int
- foldl (op =>);
val it = fn : int -> int list -> int
- foldl (op => @);
val it = fn : int list -> int
-
```

CS216

68

foldr Vs foldl

```
# Standard ML of New Jersey 110.0.7
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 (CM&CMB)
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr (op =>) @ [1,2,3,4];
val it = 2 : int
- foldl (op -> @ [1,2,3,4]);
val it = 2 : int
```

CS216

69

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
-
```

CS216

70