Functional Programming

Imperative versus Functional Languages

attributes of imperative languages:

variables, destructive assignment, sequential execution,
program semantics reflect computer architecture,
loops are important

attributes of functional languages:

data objects and mathematical functions,
programs without side-effects $\longrightarrow$ referential transparency,
recursion and functional forms instead of loops

Functions

function maps **domain** into **range**

signature of function determines domain and range:

square: integer $\longrightarrow$ natural

mapping rule specifies value of range associated with value of domain:

square(n) $\equiv$ n $\times$ n

application of function to value returns result:

square(2)

variables in functional languages differ from those in imperative languages:
single assignment

Functional Forms

functional forms combine functions to new functions:

$$F \equiv G \circ H \qquad \text{(composition)}$$

functional form replace sequential execution

examples of recursive functions:

$$n! \equiv \text{if } n = 0 \text{ then } 1 \text{ else } n * (n\text{-}1)!$$

$$\text{prime}(n) \equiv \text{if } n = 2 \text{ then true else } p(n, \text{sqrt}(n))$$

$$p(n, i) \equiv \text{if } (n \bmod i) = 0 \text{ then false}$$
$$\text{else if } i = 2 \text{ then true}$$
$$\text{else } p(n, i - 1)$$

Values, Bindings, Functions

bindings in ML:

```
val A = 3;
val B = "a";
```

functions are values:

```
val sq = fn(x:int) => x * x;
fun square(n:int) = n * n;
```

function applications:

```
(fn(x:int) => x * x) 2
2 * sq(A)
```

Factorial in C++ and ML

```
int fact(int n) {
    int i = 1;
    for (int j = n; j > 1; j--)
        i = i * j;
    return i;
}
```

```
int fact(int n) {
    if (n == 0)
        return 1;
    return n * fact(n - 1);
}
```

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

Expressions in the Lambda Calculus

expressions in the lambda calculus are defined as:

names (identifiers): x, 3, *

function abstractions (λx.e) where x is a name (formal parameter)
and e an expression (body of function), e.g., (λx.x $*$ x)

function applications (f e) where expression f is applied to expression e,
e.g., ((λx.x $*$ x) 2)

parentheses can be omitted: e1 e2 e3 = ((e1 e2) e3)
λx.y z = (λx.(y z))

Currying

usual notation f(x) in lambda-calculus replaced by f x
(parentheses are optional)

however, f(x,y,z) cannot be directly expressed,
rather use nested functions (**currying**)

example: $(\lambda x.\lambda y.\lambda z.x * y * z)\ 1\ 2\ 3$

with parentheses and stepwise reduction:

$$((((\lambda x.(\lambda y.(\lambda z.(x * y * z))))\ 1)\ 2)\ 3)$$
$$(((\lambda y.(\lambda z.(1 * y * z)))\ 2)\ 3)$$
$$((\lambda z.(1 * 2 * z))\ 3)$$
$$(1 * 2 * 3)$$

Substitution

e1 replaces each free (unbound) occurrence of x in e2, formally [e1/x]e2

$$
\begin{aligned}
[e/x1]x2 &= e \quad \text{if} \quad x1 = x2 \\
&= x2 \quad \text{if} \quad x1 \neq x2
\end{aligned}
$$

$$
\begin{aligned}
[e1/x1](\lambda x2.e2) &= (\lambda x2.e2) \quad \text{if} \quad x1 = x2 \\
&= (\lambda x2.[e1/x1]e2) \quad \text{if} \quad x1 \neq x2 \quad \text{and} \\
&\qquad\qquad x2 \text{ does not occur free in e1} \\
&= (\lambda x3.[e1/x1][x3/x2]e2) \quad \text{otherwise,} \\
&\qquad\qquad \text{where} \quad x1 \neq x3 \neq x2 \quad \text{and} \\
&\qquad\qquad x3 \text{ does not occur free in e1 and e2}
\end{aligned}
$$

$$[e1/x](e2\ e3) = ([e1/x]e2\ [e1/x]e3)$$

Equivalence of Lambda Expressions

two expressions are equivalent if they can become equal by repeatedly applying these nondirectional **conversion rules**

renaming of parameters (α conversion):

λx1.e $\leftrightarrow$ λx2.[x2/x1]e (where x2 is not frei in e)

function application (β conversion):

(λx.e1) e2 $\leftrightarrow$ [e2/x]e1

elimination of redundant functions (η conversion):

λx.(e x) $\leftrightarrow$ e (where x is not free in e)

Reduction and Normal Form

reduction rules correspond to the conversion rules,
but the β rule and the η rule are applicable only from left to right

an expression is in **normal form** if neither β nor η reduction is applicable

not each lambda expression has a normal form

all repeated applications of reduction rules to a lambda expression that lead to a normal form result in the same normal form (up to α conversion)

Example of Reduction

example of reduction to normal form:

$$
\begin{aligned}
&(\lambda x.(\lambda y.x + y)\ 5)\ ((\lambda y.y * y)\ 6) = \\
&(\lambda x.x + 5)\ ((\lambda y.y * y)\ 6) = \\
&(\lambda x.x + 5)\ (6 * 6) = \\
&(6 * 6) + 5
\end{aligned}
$$

the lambda calculus itself does not handle primitive operations like $+$ and $*$,
but variants of the calculus provide such extensions

there are typed variants of the lambda calculus

- some with similar properties as the untyped calculus,
- but also some where each expression has a normal form (quite restrictive)

LISP

examples of symbols:	A	(atom)
	AUSTRIA	(atom)
	68000	(number)

examples of lists: (PLUS A B)
((MEAT CHICKEN) WATER)
(UNC TRW SYNAPSE RIDGE HP)
NIL and () represent the empty list

a number represents its value directly,
an atom stands for another value (similar to variable name)

(SETQ X (A B C)) binds X globally to the result of (A B C),
(LET ((X A) (Y B)) E) binds X locally in E to A and Y to B

Primitive Functions in LISP

the first element of a list is usually interpreted as a function
applied to the further list elements

(QUOTE A), as well as 'A for short, returns A itself
without computing the value of A

examples of primitive functions:

(CAR '(A B C)) = A
(CAR 'A) = error
(CDR '(A B C)) = (B C)
(CDR '(A)) = () = NIL
(CONS 'A '(B C)) = (A B C)
(CONS '(A) '(B)) = ((A) B)

(ATOM 'A) = T
(ATOM '(A)) = NIL
(EQ 'A 'A) = T
(EQ 'A 'B) = NIL
(COND ((EQ 'X 'Y) 'B) (T 'C)) = C
(NULL '()) = T

Functions in LISP

(LAMBDA (X Y) (PLUS X Y)) is function with two parameters

((LAMBDA (X Y) (PLUS X Y)) 2 3) uses the function and returns 5 as result

(DEFINE (ADD (LAMBDA (X Y) (PLUS X Y)))) defines global function name

```
(DEFINE (REVERSE (LAMBDA (L) (REV NIL L))))
(DEFINE (REV (LAMBDA (OUT IN)
        (COND ((NULL IN) OUT)
              (T (REV (CONS (CAR IN) OUT)
                      (CDR IN)))))))
```

Properties of LISP

well-known functional form: (MAPCAR SQUARE L)

applies SQUARE to each element of list L, returns list of results

it is easy to define new functional forms

LISP is syntactically and semantically very simple,
a LISP interpreter can be written in a small piece of code

there are libraries with a huge number of functions and functional forms usable in many application areas; therfore, LISP is still very popular in some communities

APL

APL programmers frequently use destructive assignments,
nonetheless APL is essentially an applicative functional language

data objects are “scalars” (numbers and characters) and “arrays” (sequences of elements separated by white space), 1 and 0 represent true and false

examples of assignments:

$$X \leftarrow 123$$
$$X \leftarrow \text{'b'}$$
$$X \leftarrow 5\ 6\ 7\ 8\ 9$$

assignments produce values:

$$X \leftarrow (Y \leftarrow 5\ 6\ 7\ 8\ 9) \times (Z \leftarrow 9\ 9\ 7\ 6\ 5)$$
$$W \leftarrow Y - Z$$

Functions in APL

there are about hundred primitive functions (operations):

arithmetic operations: $+$, $-$, $\times$, $\div$, $|$ (rest of division)
Boolean operations: $\wedge$, $\vee$, $\sim$, $<$, $\leq$, $=$, $\geq$, $>$, $\neq$

all scalar operations are (element by element) applicable to arrays

"ι" generates arrays: $\iota 5 \equiv 1\ 2\ 3\ 4\ 5$

";" concatenates two arrays: $\iota 4; \iota 5 \equiv 1\ 2\ 3\ 4\ 1\ 2\ 3\ 4\ 5$

"ρ" arranges arrays: $2\ 3\ \rho\ 1\ 2\ 3\ 4\ 5\ 6 \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

"$/$" condenses arrays: $1\ 0\ 0\ 1\ /\ \iota 4 \equiv 1\ 4$

Functional Forms in APL

the reduction operator "/" applies an operation step by step to an array:

$+/(\iota 3) \equiv 6$
$+/(2\ 3\ \rho\ \iota 6) \equiv +/[2](2\ 3\ \rho\ \iota 6) \equiv 6\ 15$
$+/[1](2\ 3\ \rho\ \iota 6) \equiv 5\ 7\ 9$

the inner product operator "." used as X f.g Y on matrices X and Y
applies g element by element to X (line by line) and Y (column by column)
and finally performs a reduction using f

example: X $+.\times$ Y multiplies the matrices X and Y

the outer product operator "$\circ$" used as A $\circ$.f B on arrays A and B
applies f to each pair of elements from A and B:

example: $1\ 2 \circ.\times 3\ 4\ 5 \equiv \begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix}$

A Program in APL

computation of all prime numbers from 1 to N

step 1: $(\iota N) \circ.| (\iota N)$

step 2: $0 = (\iota N) \circ.| (\iota N)$

step 3: $+/[2]\ 0 = (\iota N) \circ.| (\iota N)$

step 4: $2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N))$

step 5: $(2 = (+/[2]\ 0 = (\iota N) \circ.| (\iota N))) / \iota N$

ML

lists: [2, 3, 4]
["a", "b", "c"]
[true, false]
[] is equivalent to nil

list operations: hd([1, 2, 3]) ≡ 1
tl([1, 2, 3]) ≡ [2, 3]
1::[2, 3] ≡ [1, 2, 3]
[1, 2]@[3] ≡ [1, 2, 3]

local bindings:

```
let val x = 5 in 2 * x * x end;

local val x = 5 in val sq = x * x;
                   val cube = x * x * x
                end;
```

Simple Programs in ML

```
fun reverse([]) = []
|   reverse(x::xs) = reverse(xs)@[x];

fun insert(x, []) = [x]
|   insert(x:int, y::ys) =
        if x < y then x::y::ys
                 else y::insert(x, ys);

fun sort([]) = []
|   sort(s::xs) = insert(s, sort(xs));
```

Functions in ML

anonymous functions: `fn(x:int) => -x;`
`fn(x:int, y:int) => x * y;`

use: `val intnegate = fn(x:int) => -x;`

type: `fn:int->int`

higher-order function: `fun compose(f,g) = f(g(x));`

type: `fn:('a->'b * 'c->'a)->('c->'b)`

curried function: `fun times(x:int)(y:int) = x * y;`

type: `fn:int->(int->int)`

use: `fun multby5 = times(5);`

Functional Forms in ML

map corresponds to MAPCAR in LISP; map is curried:

map length [[], [1, 2, 3], [3]] ≡ [0, 3, 1]

this function corresponds to the reduction operator / in APL:

```
fun fold(f, s, nil) = s
|   fold(f, s, (h::t)) = f(h, fold(f, s, t));
```

use: fold((op+), 0, [1,2,3,4]) ≡ 10

functional forms can be realized as higher-order functions using currying

Types in ML

predefined types: bool, int, real, string

lists:
[1, 2, 3] : int list
["a", "b"] : string list
nil : 'a list

tupels:
(true, "fact", 7) : bool * string * int
(true, nil) : bool * ('a list)

records: {name = "A", id = 9} : {name:string, id:int}

functions: idfunction : 'a->'a

type definitions:
type intpair = int * int;
type 'a pair = 'a * 'a;
type boolpair = bool pair;

Data in ML

datatype color = red | green | blue;

datatype publications = nopubs | journal of int | conf of int;

datatype 't stack = empty | push of 't * 't stack;

values of stack: empty
push(2, empty)
push(2, push(3, push(4, empty)))

datatype 't list = nil | :: of 't * 't list;

Abstract Data Type in ML

```
abstype 'a lifo = Stack of 'a list
with  exception error;

      val create = Stack nil;

      fun push(x, Stack xs) = Stack(x::xs);

      fun pop(Stack nil) = raise error
      |   pop(Stack(x::xs)) = Stack xs;

      fun top(Stack nil) = raise error
      |   top(Stack(x::xs)) = x;

      fun length(Stack nil) = 0
      |   length(Stack(x::xs)) = length(Stack xs) + 1;
end;
```

Module in ML

```
structure S = struct
    exception error;
    datatype 't Stack = 't list;

    val create = Stack nil;

    fun push(x, Stack xs) = Stack(x::xs);

    fun pop(Stack nil) = raise error
    |   pop(Stack(x::xs)) = Stack xs;

    fun top(Stack nil) = raise error
    |   top(Stack(x::xs)) = x;

    fun length(Stack nil) = 0
    |   length(Stack(x::xs)) = length(Stack xs) + 1;
end;
```

Signature in ML

```
signature stringStack = sig
    exception error;
    type string Stack;

    val create: string Stack;
    val push: string * string Stack -> string Stack;
    val pop: string Stack -> string Stack;
    val top: string Stack -> string;
end;
```