7. Other Paradigms

Scripting Languages, Functional Programming, Logic Programming
Scripting Languages

- Categories
- Characteristics
- Some languages
- Sample Python script
Categories

- Shell scripts
  - Also called "batch" or "command" scripts
- Text-processing languages
- "Glue" languages
  - Put together several different programs
- Web scripts
- Extension languages
  - Allow new commands to be added to applications
Characteristics

- Can be used either interactively or as programs
- Avoid extensive syntax
- Have no declarations and simple rules
- Use dynamic typing
- Provide easy access to system and other programs
- Perform pattern-matching and string manipulation
- Include built-in high-level data types
Some Languages

- Python
  - General-purpose, command scripting, "glue"

- Perl
  - General-purpose, command scripting, "glue", text processing, server-side web scripting

- Ruby
  - Like Python, but strongly object-oriented, and Perl-like

- JavaScript
  - Client-side web scripting, extensions

- Visual Basic for Applications
  - Extensions
```python
import glob
import subprocess

JAVA_PATH = 'C:/Program Files/Java/jdk1.6/bin'
PROGRAM_FUNCTION = 'public static void main(String[] args)'

print 'Each Java program will be compiled and executed, then listed.'
java_files = glob.glob('*.java')
for filename in java_files:
    current_file = open(filename, 'r')
    is_program = False
    for line in current_file:
        if PROGRAM_FUNCTION in line:
            is_program = True
            break
    if is_program:
        print '=============================================
        class_name = filename[5:-5]
        print '----------------------'
        print 'Compiling', class_name
        subprocess.call(JAVA_PATH + '/javac ' + filename);
        print '----------------------'
        print 'Executing', class_name
        subprocess.call(JAVA_PATH + '/java ' + class_name)
        print 'Execution ends'
        print '----------------------'
        subprocess.call('pause', shell = True)
        print
```
Functional Programming

- Characteristics
- Conditional expressions
- Recursion
- Side effects
- Advantages
- Lists
- Functions are first-class
Characteristics

- Execution is controlled through conditionals and recursion
- No side effects are allowed
- Lists are the principal data structure
- Functions are *first class*
Value of an expression depends on another, Boolean-valued, expression

Similar to ?: operator in C/C++/Java

Example (using C/C++/Java syntax)

```c
int y = 10;
int x = y > 5 ? 8 : 12;
```

In these languages, more typical to use conditional statements

```c
int y = 10; int x;
if (y > 5) x = 8; else x = 12;
```

Example (using Haskell syntax)

```
x = if y > 5 then 8 else 12 where y = 10
```
 Nested Conditionals

- **C/C++/Java statement**
  ```
  int y = 10;
  if (y==0)
      x = 1;
  else if (y==1)
      x = 2;
  else if (y==2)
      x = 3;
  else
      x = 11;
  ```

- **C/C++/Java expression**
  ```
  x = y==0 ? 1 : y==1 ? 2 : y==2 ? 3 : 11;
  ```

- **Haskell expression**
  ```
  let y = 10 in if y==0 then 1 else if y==1 then 2 else if y == 2 then 3 else 11
  ```
Multi-way Conditionals

- C/C++/Java statement (no multi-way expression)
  ```java
  int y = 10;
  switch (y)
  {
    case 0:
      x = 1; break;
    case 1:
      x = 2; break;
    case 2:
      x = 3; break;
    default:
      x = 11;
  }
  ```

- Haskell expression
  ```haskell
  let y = 10 in case y of 0->1; 1->2; 2->3; otherwise->11
  ```
Recursion

- Same mechanism in most languages
  - Function calls itself, directly or indirectly (with escape)
- Used much more extensively in functional languages
  - Only mechanism for repetition
  - Not disrespected as a source of inefficiency
    - Compiler can deal with inefficiencies of implementation
    - Logic can cure poor programming practice
Example: Factorial in Haskell

```haskell
-- Usual recursive definition
factorial1 0 = 1
factorial1 n = n * factorial1 (n - 1)
```

```haskell
-- Better recursive definition
-- undefined for n < 0
factorial2 n
  | n == 0 = 1
  | n >= 1 = n * factorial2 (n - 1)
```

```haskell
-- "Iterative" definition (Tail recursion)
factorial3 n = helper 1 1
  where helper accum count
    | count > n = accum
    | otherwise = helper (count * accum) (count + 1)
```
--- Recursive, but very inefficient
--- Fault of algorithm, not recursion itself
fibonacci1 1 = 1
fibonacci1 2 = 1
fibonacci1 n = previous + oneBeforeThat
   where
   previous = fibonacci1 (n - 1)
   oneBeforeThat = fibonacci1 (n - 2)

--- Tail recursive - and efficient
fibonacci2 n = helper 1 1 1
   where helper previous current count
       | count == n = current
       | otherwise = helper (previous + current)
                      previous
                      (count + 1)
Side Effects

- Any direct change in state – brought about by changing the value of a variable
- Example in C++ and Java (not Haskell)

```cpp
#include <iostream>
using namespace std;

int x = 0;
int f(int n)
{
    ++x;  //side effect
    return n + x;
}

int main()
{
    for (int i = 0; i < 3; ++i)
        cout << f(2) << endl;
    return 0;
}
```

```java
public class SideEffects
{
    public static int x = 0;
    public static int f(int n)
    {
        ++x;  //side effect
        return n + x;
    }
    public static void main(String[] args)
    {
        for (int i = 0; i < 3; ++i)
            System.out.println( f(2) );
    }
}
```
Most programmers consider side effects harmful when a function modifies a global variable.

But class methods modify instance variables in exactly this way.

And loops with counters or accumulators modify variables to achieve results.

Functional language programmers consider all side effects harmful.

Pure functional languages disallow assignment – initialization only. No variables – just constants!
Accumulators, counters, etc., in loops replaced by use of *recursion*
- “Iterative” (tail-) recursion allows for efficiency
- Some functional languages provide *syntactic sugar* to introduce loop syntax (but it’s still recursion)

Lists and list operations take place of many iterative operations on arrays
- However, lists cannot be changed – operations return whole new list
- Interpreter or compiler can often remove such inefficiencies
Advantages

- Functions act as they do in mathematics
  - No “assignments” in math
  - Well-developed theory
    - It’s math(!)
    - Lambda calculus
- Parameter-passing method is irrelevant
  - Only mode is input mode
  - No side effects allowed
  - So pass by value, reference, name, and need all produce same result
  - Lazy evaluation also gives same result
Lists

- Lists are built in
- Methods exist to
  - Pick off pieces from list
  - Build a list from its pieces
- Functions operating on individual items can be “mapped” over entire lists
  - Map method itself can be written using recursion on lists
Examples

> [1,2,3,4,5]
[1,2,3,4,5]
> [1..5]
[1,2,3,4,5]
> [1..5] !! 2
3
> take 2 [1..5]
[1,2]
> head [1..5]
1
> tail [1..5]
[2,3,4,5]

> head (tail [1..5])
2
> head (head [[1..5], [7..12], [15..20]])
1
> head (tail [[1..5], [7..12], [15..20]])
[7,8,9,10,11,12]
> head ( head (tail [[1..5], [7..12], [15..20]]))
7
> filter even [1..20]
[2,4,6,8,10,12,14,16,18,20]
> [x | x <- [1..20], even x]
[2,4,6,8,10,12,14,16,18,20]
> toUpper 'a'
 'A'

> toUpper "abcde"
ERROR - Type error in application
*** Expression : toUpper "abcde"
*** Term : "abcde"
*** Type : String
*** Does not match : Char

> map toUpper ['a', 'b', 'c', 'd', 'e']
"ABCDE"

> map toUpper "abcde"
"ABCDE"

> map oddify [1..10] where oddify x = 2*x + 1
[3,5,7,9,11,13,15,17,19,21]

> map (\x -> 2*x + 1) [1..10]
[3,5,7,9,11,13,15,17,19,21]
Functions Are First-class

- Functions can
  - Be used as values, e.g., to initialize variables
  - Be included in lists
  - Be passed as parameters
  - Be returned as values

- They have all the “rights” of other objects
Lisp and Scheme

- Lisp was the original functional language.
- It was, in fact, one of the first programming languages of any kind – only Fortran is older.
- Still used in AI and some other applications.
- Was not used here to introduce functional languages because its syntax is rather odd, and gets in the way of seeing the important points of functional style.
- We will look at the dialect of Lisp called Scheme.
Symbols and Values

- Everything in Scheme is either an atom or a pair (which is, as far as we are concerned, just a strange way to say list)
- An atom is either a number or a symbol
  - A symbol can be used as a name (of a variable or function)
    - You do this with a definition, such as (define a 2)
  - A symbol can be quoted, in which case it acts similarly to (not the same as) a string
    - Use 'a instead of a
    - The character #\a or the string "a" can also be used, but neither is the same as the symbol a
  - The main operational difference from Java, etc., is that you can go back and forth between meanings
Examples

> \"a\" ; character a
\n\n> \"a\" ; string a
\n> a ; symbol a (no value defined)

⚠️ Reference to an identifier before its definition: a

> 'a ; quoted symbol (no value needed since not evaluated)

> (define a 2) ; a now defined

> a

2

> 'a

'a

> (define x \"a\")

> (define y \"a\")

> (define z 'a)

> (define w a)

> x

\"a\"

> y

\"a\"

> z

'a

> w

2

> (eval x)

\"a\"

> (eval y)

\"a\"

> (eval z)

2

> (eval w)

2
A list is written between parentheses, with no other punctuation

- *Example:* \((a \ b \ c)\)

A list is interpreted as a *function call* – the first item is the function, the rest are parameters

- *Example:* \((a \ b \ c)\) would be interpreted as what would be written in Java or C as \(a(b, c)\)
- Note that *prefix notation* is used

You may *quote* a list, in which case it is not evaluated, and is just treated as a list of symbols
Taking lists apart

- **Function car** returns the **head** of the list, i.e., the first item on the list
  - Example: The head of (a b c d) is a

- **Function cdr** returns the **tail** of the list, i.e., all of the list except for the first item
  - Example: the tail of (a b c d) is (b c d)

Building lists

- **Function cons** constructs a list with a given head and tail
  - Example: the list constructed from a and (b c d) is (a b c d)
> `(1 2 3)  
procedure application: expected procedure, given: 1; arguments were: 2 3  
> '1 2 3  
'(1 2 3)  
> (a b c)  
reference to an identifier before its definition: c  
> '(a b c)  
'(a b c)  
> (define a *)  
> (define b 2)  
> (define c 3)  
> (a b c)  
6  
> '(a b c)  
'(a b c)  
> (eval '(a b c))  
6  
> (define my-list '(a b c d))  
> my-list  
'(a b c d)  
> (car my-list)  
'a  
> (cdr my-list)  
'(b c d)  
> (car (cdr my-list))  
'b  
> (cadr my-list)  
'b  
> (caddr my-list)  
'c  
> (cons 'x my-list)  
'(x a b c d)  
> (cons my-list my-list)  
'((a b c d) a b c d)  
> (append my-list my-list)  
'(a b c d a b c d)
Everything in Lisp uses the same syntax: atoms are just written, and all other expressions are written as lists

This includes definitions, conditionals, ... everything

Prefix notation is used throughout with minimal use of keywords

Some say Lisp has no syntax (other than parentheses), but that is not really true

- Not everything can be a true function, all of whose arguments are evaluated, e.g., in (if a b c) not both b and c should be evaluated
- So if is a “special form” – otherwise known as syntax!
Examples

> (define x 10)
> (define y 20)
> (if (< x y) x y)
10
> (cond
    ((< x 5) 100)
    ((< y 10) 200)
    ( (> x 10) 300)
    ( (> y 30) 400)
    (else 500))
500
> (define (square x) (* x x))
> square
#<procedure:square>
> (square 5)
25
> (define (larger x y) (if (> x y) x y))
> (larger 10 20)
20
> (define second cadr)
> (second '(a b c))
'b
> (define 3rd caddr)
> (3rd '(a b c))
'c
Example: Factorial in Scheme

; Usual recursive definition
(define (factorial1 n)
  (if (= n 0) 1 (* n (factorial1 (- n 1))))))

; Better recursive definition
; Error for n < 0
(define (factorial2 n)
  (cond
    ((< n 0) 'ERROR)
    ((= n 0) 1)
    (else (* n (factorial2 (- n 1)))))))

; "Iterative" definition (tail recursion)
(define (factorial3 n)
  (letrec ((helper
    (lambda (accum count)
      (if (> count n)
          accum
          (helper (* count accum) (+ count 1)))))))
  (helper 1 1))
Example: Fibonacci in Scheme

; Recursive, but very inefficient
; Fault of algorithm, not recursion itself
(define (fibonacci1 n)
  (cond
    ((= n 1) 1)
    ((= n 2) 1)
    (else
      (letrec ((previous (fibonacci1 (- n 1)))
                  (one-before-that (fibonacci1 (- n 2))))
        (+ previous one-before-that)))))

; Tail recursive -- and efficient
(define (fibonacci2 n)
  (letrec ((helper
                (lambda (previous current count)
                  (if (= count n)
                      current
                      (helper (+ previous current) previous (+ count 1))))))
    (helper 1 1 1)))
More Examples

```scheme
(define (to-upper ch)
  (let* ((a (char->integer #\a))
         (z (char->integer #\z))
         (c (char->integer ch))
         (letter? (and (>= c a) (<= c z)))
         (if letter? (integer->char (– c 32)) ch)))

> (to-upper #\a)
#\A
> (to-upper "abc")
char->integer: expects argument of type <character>; given "abc"

> (define (to-upper-case str)
  (list->string (map to-upper (string->list str)))

> (to-upper-case "abc")
"ABC"

> (map (lambda (x) (+ (* 2 x) 1)) '(1 2 3 4 5 6 7 8 9 10))
'(3 5 7 9 11 13 15 17 19 21)
```
All programming in form of logic: axioms and theorems to prove.
- Axioms are either facts or rules
- Theorems to prove are queries

Non-procedural
- You state a fact or rule and query
- You don’t say how

Incomplete
- Not all computations are possible
Characteristics

- Uses algorithms matching facts and rules with queries and their parts to decide queries
  - Resolution principle
  - Unification
- Prolog
  An implemented logic programming language (*programmation en logique*)
In logic
- \( P \lor Q, \neg Q \lor R \mid - P \lor R \)

In English
- Given
  - Either \( P \) or \( Q \) is true, and
  - Either \( Q \) is false or \( R \) is true
- Conclude
  - Either \( P \) is true or \( R \) is true
Basic idea is substitution
- Notation: $x/y$ means “replace $x$ by $y$”

Sometimes you have a chain of substitutions
- $x/y$ then $y/z$, yielding $x/z$

Relatively easy for human beings, but need a careful definition to develop an algorithm

Although idea is simple, many cases make careful definition complicated
Combination of resolution and unification allows simplification

**Example**

- \( \forall x (\neg P(x) \lor Q(x)), \forall y (\neg Q(y) \lor R(y)) \)
- “Unify”:
  - \( \forall x (\neg P(x) \lor Q(x)), \forall x (\neg Q(x) \lor R(x)) \)
  - \( \forall x (\neg P(x) \lor Q(x), \neg Q(x) \lor R(x)) \)
  - Resolve: \( \forall x (\neg P(x) \lor R(x)) \)
**Prolog**

- *Prolog* is the principle language using logic programming
- Syntax varies among implementations. Typical:
  - `predicate(param₁, ... paramₙ).` (fact)
  - `pred :- fact₁,...,factₙ.` (rule: *pred if fact₁ and ...and factₙ*)
  - `?- predicate(param₁, ... paramₙ).` (query)
  - Identifier starting with uppercase letter is a *variable*, and one starting with lowercase letter is a *constant*
female(carla).
female(joan).
female(marge).
female(angie).
female(janet).
female(kate).
female(jessica).
female(anne).
female(isabella).
female(lorraine).
female(michelle).

male(john).
male(frank).
male(dan).
male(joe).
male(mike).
male(tony).
male(steve).
male(jerry).
male(george).
male(ed).

parents(carla, john, joe).
parents(carla, john, mike).
parents(joan, frank, marge).
parents(joan, frank, angie).
parents(janet, dan, george).
parents(janet, dan, kate).
parents(marge, joe, jessica).
parents(marge, joe, anne).
parents(kate, mike, isabella).
parents(kate, mike, tony).
parents(angi, steve, jerry).
parents(angi, steve, lorraine).
parents(lisa, dan, ed).
parents(lisa, dan, michelle).

mother(X,Y) :- parents(X,_,Y).
father(X,Y) :- parents(_,X,Y).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

child(X,Y) :- parent(Y,X).
daughter(X,Y) :- female(X), parent(Y,X).
son(X,Y) :- male(X), parent(Y,X).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
grandfather(X,Y) :- male(X), grandparent(X,Y).
grandmother(X,Y) :- female(X), grandparent(X,Y).

grandchild(X,Y) :- grandparent(Y,X).
granddaughter(X,Y) :- female(X), grandchild(X,Y).
grandson(X,Y) :- male(X), grandchild(X,Y).

sibling(X,Y) :- parents(U,V,X), parents(U,V,Y). X \= Y.
brother(X,Y) :- male(X), sibling(X,Y).
sister(X,Y) :- female(X), sibling(X,Y).

cousin(X,Y) :- parent(U,X), parent(V,Y), sibling(U,V), \+sibling(X,Y).
Implementation is not “pure”
- Searches facts in same order they are entered by programmer
- Uses negation as failure
- Allows use of procedural ideas, such as fail predicate, cut, and retract “query”

Simple things can be difficult if generally thought of procedurally
- Example: definition of a function
Examples

smart(harry).
smart(tom).
smart(pat).

smarter(X,Y) :- smart(X), stupid(Y).

rainy(today).
rainy(tom).

rainy(yesterday).

nice(\today) :- sunny(\today), \+ rainy(\today).

strange(\today) :- sunny(\today), foggy(\today).

disgusting(\today) :- rainy(\today), foggy(\today).

factorial(0,1) :- !.
factorial(X,Y) :- Z is X - 1, fact(Z,W), Y is X*W.