PROGRAMMING PARADIGMS

Programming Paradigms

- During the next few weeks we are going to work with functional programming. Before I can explain to you what FP is, I thought I'd better put things into perspective by talking about other <mark>programming paradigms</mark>.

- Over the last 40 or so years, a number of programming paradigms (a programming paradigm is a way to think about programs and programming) have emerged.

Programming Paradigms…

A <mark>programming paradigm</mark>

- is a way to think about programs, programming, and problem solving,
- is supported by one or more programming languages.

Being familiar with several paradigms makes you a better programmer and problem solver. The most popular paradigms:

1. Imperative programming.
2. Functional programming.
3. Object-oriented programming.
4. Logic Programming.

When all you have is a hammer, everything looks like a nail.

Programming Paradigms...

Imperative Programming

- Programming with <mark>state</mark>.

- Also known as <mark>procedural programming</mark>. The first to emerge in the 1940s-50s. Still the way most people learn how to program.

- FORTRAN, Pascal, C, BASIC.

Functional Programming

- Programming with <mark>values</mark>.

- Arrived in the late 50s with the LISP language. LISP is still popular and widely used by AI people.

- LISP, Miranda, Haskell, Gofer.

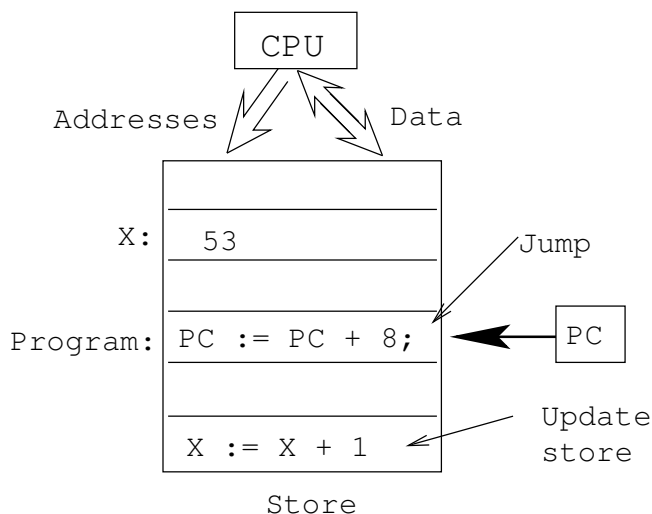# Programming Paradigms…

Object-Oriented Programming

- Programming with <mark>objects</mark> that encapsulate data and operations.

- A variant of imperative programming first introduced with the Norwegian language Simula in the mid 60s.

- Simula, Eiffel, Modula-3, C++.

Logic Programming

- Programming with <mark>relations</mark>.

- Introduced in the early 70s. Based on predicate calculus. Prolog is popular with Computational Linguists.

- Prolog, Parlog.

Procedural Programming

We program an abstraction of the Von Neumann Machine, consisting of a store (memory), a program (kept in the store), A CPU and a program counter (PC):

```
                CPU

Addresses              Data


        X:   53                  Jump

Program:  PC := PC + 8;   ◄──  PC

                                Update
          X := X + 1            store
            Store
```

Computing `x:=x+1`

1. Compute `x`'s address, send it to the store, get `x`'s value back.

2. Add 1 to `x`'s value.

3. Send `x`'s address and new value to the store for storage.

4. Increment `PC`.

Procedural Programming…

- uses <mark>control structures</mark> (IF, WHILE, ...) to alter the program counter (PC),

- uses <mark>assignment statements</mark> to alter the store.

- is in charge of <mark>memory management</mark>, i.e. declaring variables to hold values during the computation.

```
function fact (n:integer):integer;
var s,i : integer := 1;
begin
    while i<=n do s:=s*i; i:=i+1; end;
    return s;
end fact.
```

Procedural Programming…

Procedural programming is difficult because:

1. A procedural program can be in a large number of states. (Any combination of variable values and PC locations constitutes a possible state.) The programmer has to keep track of all of them.

2. Any global variable can be changed from any location in the program. (This is particularly true of languages like C & C++ [Why?]).

3. It is difficult to reason mathematically about a procedural program.

Functional Programming

In contrast to procedural languages, functional programs don't concern themselves with state and memory locations. Instead, they work exclusively with values, and expressions and functions which compute values.

- Functional programming is not tied to the von Neumann machine.

- It is not necessary to know anything about the underlying hardware when writing a functional program, the way you do when writing an imperative program.

- Functional programs are more declarative than procedural ones; i.e. they describe what is to be computed rather than how it should be computed.

Functional Languages

Common characteristics of functional programming languages:

1. Simple and <mark>concise syntax</mark> and semantics.

2. Repetition is expressed as <mark>recursion</mark> rather than iteration.

3. <mark>Functions are first class objects</mark>. I.e. functions can be manipulated just as easily as integers, floats, etc. in other languages.

4. <mark>Data as functions</mark>. I.e. we can build a function on the fly and then execute it. (Some languages).

Functional Languages…

5. Higher-order functions. I.e. functions can take functions as arguments and return functions as results.

6. Lazy evaluation. Expressions are evaluated only when needed. This allows us to build infinite data structures, where only the parts we need are actually constructed. (Some languages).

7. Garbage Collection. Dynamic memory that is no longer needed is automatically reclaimed by the system. GC is also available in some imperative languages (Modula-3, Eiffel) but not in others (C, C++, Pascal).
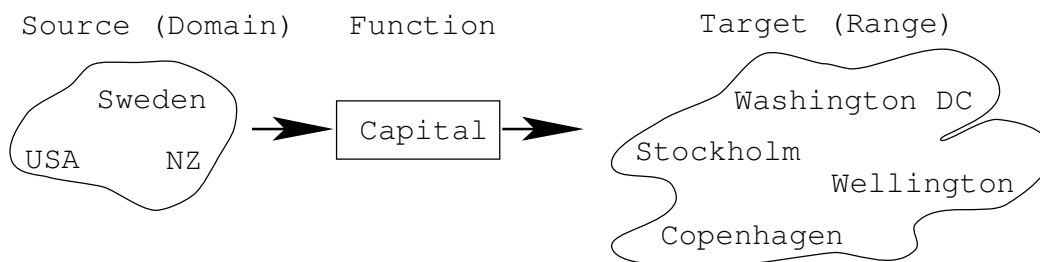
Functional Languages…

8. Polymorphic types. Functions can work on data of different types. (Some languages).

9. Functional programs can be more easily manipulated mathematically than procedural programs.

<center>Pure vs. Impure FPL</center>

- Some functional languages are pure, i.e. they contain no imperative features at all. Examples: Haskell, Miranda, Gofer.

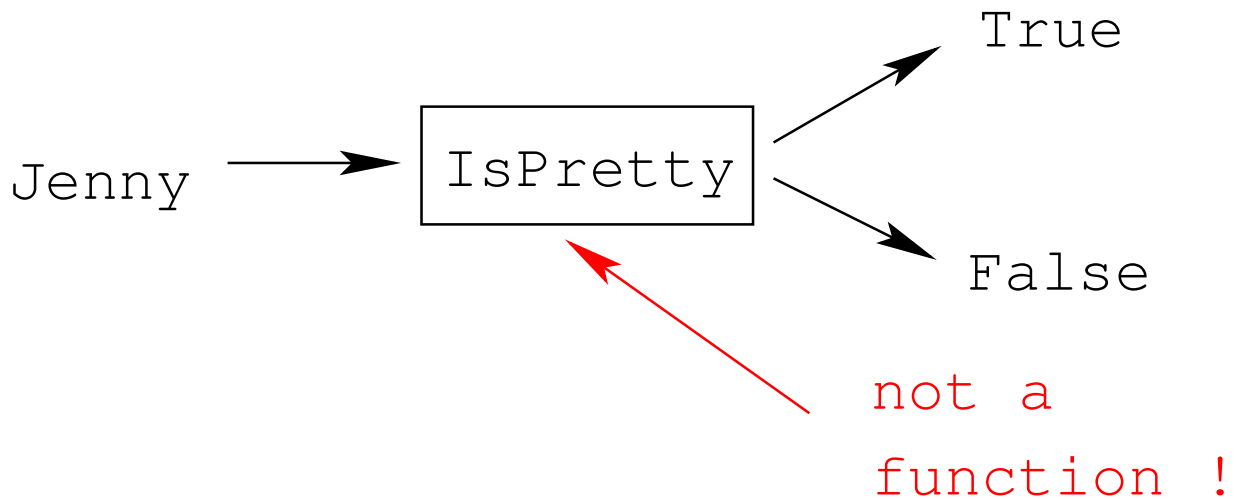- Impure languages may have assignment-statements, goto:s, while-loops, etc. Examples: LISP, ML, Scheme.

What is a function?

- A function <mark>maps</mark> argument values (inputs) to result values (outputs).

- A function takes argument values from a <mark>source set</mark> (or <mark>domain</mark>).

- A function produces result values that lie in a <mark>target set</mark> (or <mark>range</mark>).

```
Source (Domain)    Function            Target (Range)

      Sweden                            Washington DC
                   ┌─────────┐
   USA      NZ     │ Capital │      Stockholm
                   └─────────┘                 Wellington

                                          Copenhagen
```

More on functions

- A function must not map an input value to <mark>more than one</mark> output value. Example:

```
                                                   True

Jenny  ────────▶  │ IsPretty │
                                                   False

                         not a
                         function !
```

More on functions...

- If a function F maps every element in the domain to some element in the range, then F is <mark>total</mark>. I.e. a total function is defined for all arguments.

```
Int  ──────▶ ┌─────────┐
             │  Plus   │ ──────▶  Int
Int  ──────▶ └─────────┘
```

More on functions...

- A function that is undefined for some inputs, is called <mark>partial</mark>.

$$\text{Int} \searrow$$
$$\boxed{\text{Divide}} \longrightarrow \text{Int}$$
$$\text{Int} \nearrow$$

- `Divide` is partial since $\frac{?}{0} =?$ is undefined.

Specifying functions

A function can be specified <mark>extensionally</mark> or <mark>intentionally</mark>.
Extensionally:

- Enumerate the elements of the (often infinite) set of pairs "`(argument, result)`" or "`Argument ↦ Result.`"

- The extensional view emphasizes the <mark>external behavior</mark> (or <mark>specification</mark>), i.e. <mark>what</mark> the function does, rather than <mark>how</mark> it does it.

```
double = {···, (1,2), (5,10), ···}
even = {···, (0,True), (1,False), ···}
double = {···, 1↦2, 5↦10, ···}
isHandsome={Chris↦True,Hugh↦False}
```

Specifying functions…

- Give a **rule** (i.e. **algorithm**) that computes the result from the arguments.

- The intentional view emphasizes the **process** (or algorithm) that is used to compute the result from the arguments.

```
double x = 2 * x
even x = x mod 2 == 0
isHandsome x = if isBald x
          then True
          else False
```

Specifying functions...

Graphically:

- The graphical view is a notational variant of the intentional view.

eithereven
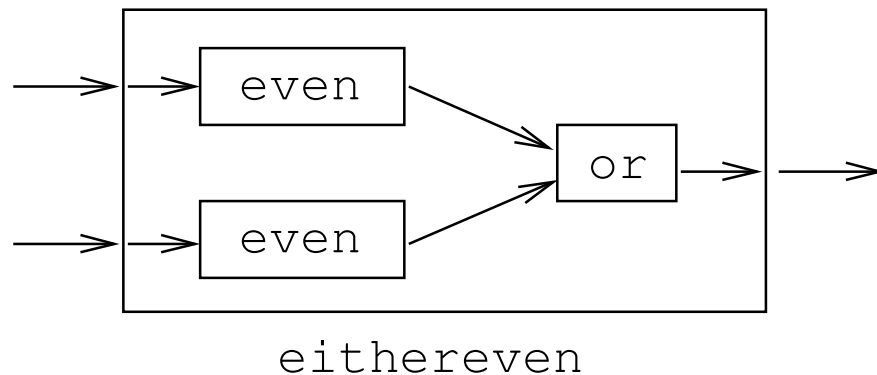
Function Application

- The most important operation in a functional program is <mark>function application</mark>, i.e. applying an input argument to the function, and retrieving the result:

```
double x = 2 * x
even x = x mod 2 == 0

double 5 ⟹ 10
even 6 ⟹ True
```

Function Composition

- **Function composition** makes the result of one function application the input to another application:

```
double x = 2 * x
even x = x mod 2 == 0

even (double 5) ⇒ even 10 ⇒ True
```