Programming Paradigms...

Imperative Programming

- Programming with state.
- Also known as procedural programming. 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 values.
- Arrived in the late 50s with the LISP language. LISP is still popular and widely used by AI people.
- LISP, Miranda, Haskell, Gofer.

Object-Oriented Programming

- Programming with objects 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 relations.
- Introduced in the early 70s. Based on predicate calculus. Prolog is popular with Computational Linguists.
- Prolog, Parlog.

A programming paradigm

- 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.”
**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.

---

**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):

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.

---

**Procedural Programming...**

The programmer...

- uses control structures (IF, WHILE, ...) to alter the program counter (PC),
- uses assignment statements to alter the store.
- is in charge of memory management, i.e. declaring variables to hold values during the computation.

```plaintext
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.
```
Common characteristics of functional programming languages:

1. Simple and **concise syntax** and semantics.
2. Repetition is expressed as **recursion** rather than iteration.
3. Functions are **first class objects**. I.e. functions can be manipulated just as easily as integers, floats, etc. in other languages.
4. **Data as functions**. I.e. we can build a function on the fly and then execute it. (Some 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 **infinit 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).
Specifying functions...

Intensionally:

- 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.

```
<table>
<thead>
<tr>
<th>even</th>
<th>or</th>
</tr>
</thead>
</table>
```

Specifying functions...

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

```
Jenny IsPretty True False
```

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

```
Int Plus Int
```

- A function that is undefined for some inputs, is called partial.

```
Int Divide Int
```

Divide is partial since $\frac{2}{0} = ?$ is undefined.

Specifying functions

A function can be specified extensionally or intentionally.

Extensionally:

- Enumerate the elements of the (often infinite) set of pairs "(argument, result)" or "Argument $\mapsto$ Result".
- The extensional view emphasizes the external behavior (or specification), i.e. what the function does, rather than how it does it.

```
double = {⋯, (1,2), (5,10), ⋯}
even = {⋯, (0,True), (1,False), ⋯}
double = {⋯, 1$\mapsto$2, 5$\mapsto$10, ⋯}
isHandsome={Chris$\mapsto$True,Hugh$\mapsto$False}
```
Function Signatures

To define a function we must specify the types of the input and output sets (domain and range, i.e. the function’s signature), and an algorithm that maps inputs to outputs.

```
int double int
  + 2
int even Bool
  mod 2 = 0
```

Application & Composition

Function Application: The most important operation in a functional program is function application, i.e. applying an input argument to the function, and retrieving the result:

- `double 5 ⇒ 10`
- `even 6 ⇒ True`

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

- `even (double 5) ⇒ even 10 ⇒ True`

Referential Transparency

- The most important concept of functional programming is referential transparency. Consider the expression

  \[(2 \times 3) + 5 \times (2 \times 3)\]

- \((2 \times 3)\) occurs twice in the expression, but it has the same meaning (6) both times.
- RT means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs.
- This concept occurs naturally in mathematics, but is broken by imperative programming languages.
- RT makes functional programs easier to reason about mathematically.

Example: How many numbers are there between \(m\) and \(n\), inclusive?

- Extensional Definition: \(\sum_{between \ m \ n} = \{\ldots (1,1) \mapsto 1, (1,2) \mapsto 2, \ldots, (2,10) \mapsto 9\}\)
- Intentional Definition: \(\sum_{between \ m \ n} = ((m + n) \times (abs (m-n) + 1)) \div 2\)
- Graphical Definition: 

```
 m + n - abs 1 + 2 \sum_{between}
```

- Referential Transparency means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs. This concept occurs naturally in mathematics, but is broken by imperative programming languages. RT makes functional programs easier to reason about mathematically.
Referential Transparency...

- RT is a fancy term for a very simple concept.
- What makes FP particularly simple, direct, and expressive, is that there is only one mechanism (function application) for communicating between subprograms.
- In imperative programs we can communicate either through procedure arguments or updates of global variables. This is hard to reason about mathematically.
- A notation (programming language) where the value of an expression depends only on the values of the sub-expressions is called referentially transparent.

Pure functional programming languages are referentially transparent. This means that it is easy to find the meaning (value) of an expression. We can evaluate it by substitution. I.e. we can replace a function application by the function definition itself.

Evaluate `even (double 5)`:

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

even (double 5) ⇒ even (2 * 5) ⇒
even 10 ⇒ 10 mod 2 == 0 ⇒
0 == 0 ⇒ True
```
Homework

- Here is a mathematical definition of the combinatorial function \( \binom{n}{r} \) “n choose r”, which computes the number of ways to pick \( r \) objects from \( n \):

\[
\binom{n}{r} = \frac{n!}{r! \cdot (n-r)!}
\]

- Give an extensional, intentional, and graphical definition of the combinatorial function, using the notations suggested in this lecture.

- You may want to start by defining an auxiliary function to compute the factorial function, \( n! = 1 \cdot 2 \cdot \ldots \cdot n \).