"The most important thing in a programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language." --- Professor Donald Knuth

Programming Paradigms

- The principal programming paradigms are:
  - Imperative (Procedural) Programming
  - Functional (Applicative) Programming
  - Object-Oriented Programming
  - Concurrent Programming
  - Logic Programming
  - Scripting Language Programming
Programming Paradigms

“Paradigms” emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas.


Programming paradigms are the result of people’s ideas about how programs should be structured/constructed and the development of formal linguistic mechanisms for expressing those ideas and software engineering principles and practices for using the resulting programming language to solve computational problems.

Imperative Programming

Imperative programs consist of actions to effect state change, principally through assignment operations.

Well-known imperative languages: C, Fortran, Pascal, Ada, Modula-2, Cobol, PL/I, Algol

Why do imperative programming languages dominate in practice?

OO programming is not always imperative programming but most OO languages have been imperative: Simula, Smalltalk, C++, Modula-3, Java.

However, CLOS (Common Lisp Object System) is an object-oriented version of Lisp that is not
Imperative Languages

Defined by a set of imperative statements/commands used to direct a computer to do something useful.

For example, in C, we have a command to print to an output device:

```c
printf("hello world\n");
```

Various kinds of type definitions and expressions

```c
int x; x = 1; x = +++x + x++;
```

Conditional statement vs conditional expression

```c
if (a < b) z = x; else z = y; or z = (a < b) ? x : y;
```

Other Paradigms

Concurrent programming cuts across imperative, object-oriented, and functional programming.

Logic programming is based on first-order logic and targeted at theorem proving languages and database applications.

Scripting is a very high-level of programming often used for rapid development

- borrows heavily from functional programming
- used to "glue" together different programs
- used to script active web pages

Monday, April 6, 2009
A Functional Point of View

A function is well-defined iff for every input there is at most one output. A function in set theory is a graph characterized solely by an input-output relation.

- **extensional equality** - two functions f & g are equal iff they have the same graph,
  \( \{(x, y) \mid y = f(x)\} \)

- This doesn't work too well in programming! In what way are two sorting functions equivalent?

- **intensional equality** - equivalent algorithmic complexity. How the function computes its result is important in CS so we also characterize a function by its algorithm:
  - e.g., a \( O(n^2) \) vs \( O(n \log n) \) sorting algorithm

Functional Languages

Defined by a more mathematical/logical syntactic notation for specifying computations using (pure) functions and expressions, independent of a specific machine.

To a functional programmer, the machine is a low-level mechanism for the execution of the abstract computations that we have specified in our high-level language notation, so state is (usually) not explicit.

For example, a factorial function takes a single non-negative integer argument and computes a positive integer result.

\[ \text{fact: } \text{nat} \rightarrow \text{nat} \]
Different Factorial Function Defns

C, C++, Java:
```c
int fact (int n) { return (n == 0) ? 1 : n * fact (n-1); }
```

Scheme:
```scheme
(define fact
 (lambda (n)
   (if (= n 0) 1 (* n (fact (- n 1))))))
```

SML:
```sml
fun fact n = if n=0 then 1 else n * fact(n-1);
```

Haskell:
```haskell
fact :: Integer->Integer
fact 0 = 1
fact n = n * fact(n-1)
```

Set Theoretic:
```
\{(x,y) | \forall i,j \in \mathbb{N}, (i,j) = y \}
```

Algebraic:
```
f : \mathbb{N} \rightarrow \mathbb{N}
\begin{align*}
f(x) &= x^2\end{align*}
```

LISP:
```
(defun square(x) (* x x))
```

Scheme:
```
(define square (lambda (x) (* x x)))
```

Algol60:
```
integer procedure square(int x); integer x;
beg square := x * x end;
```

Pascal:
```
function square (x:int) : integer;
beg square := x * x end;
```

K&R C:
```
square(x)
int x; { return (x * x); }
```

ANSI C/C++ & Java:
```
int square(int x) { return (x * x); }
```

SML97:
```
fun square x = x * x;
fun square (x:int) = x * x;
val square = fn x => x * x;
```

Haskell:
```
square :: Integer->Integer
square x = x * x
map \(\lambda x \rightarrow x \cdot x\) [1,2,3,4,5]
[(x,y) | x <- [0..], y <- [x * x]]
```
Key Functional Concepts

- polymorphic data types, recursive/inductive datatypes
- strong typing, static/dynamic typing, type inferencing
- referential transparency of expression evaluation
- lists (including infinite lists) and list comprehensions
- recursive function definitions, including “tail recursion”
- function definition using pattern matching
- “curried” functions and anonymous “lambda” functions
- higher-order functions
- strict (eager) versus non-strict (lazy) evaluation

Functional Programming

- Define a program in terms of data types, expressions and functions and (mostly) avoid side-effects
- Pure functional programming means no side effects
- side-effects most often result from the use of the assignment statement and call-by-reference parameter passing
- “Practical” functional programming languages rely on non-pure functions for input/output and some permit assignment-like operators

- Scheme: (set! x 1) “assigns” 1 to the variable x
- SML: let x = 10 in x * x; versus x := 1; x := x * x
Functional Programming History

- Alonzo Church invented a functional notation called the Lambda Calculus around 1934-35 that made precise the notion of a computable function.
- Alan Turing studied under Church for a year and defined his famous Turing Machine in 1936.
- You could legitimately argue that ALL programming languages owe some of their heritage to Church's lambda calculus since it is a logic of functions, function composition and argument evaluation.
- McCarthy's LISP was the earliest functional language followed by ISWIM, ML, FP and others.
- Interest in functional languages gained interest following John Backus' 1977 Turing Award Talk: "Can Programming be Liberated from the Von-Neumann Style? A Functional Style and Its Algebra of Programs".

Modern Functional Languages

- Haskell
  - for pure functional programming with non-strict (lazy) evaluation.
- Standard ML (SML)
  - for practical functional programming with strict (eager) evaluation & polymorphic type inferencing.
- O'Caml
  - a dialect of ML, for object-oriented functional programming.
- Scheme
  - for lexically scoped applicative-style recursive programming with latent or dynamic typing.
Why Functional Programming?

- Programmer productivity
  - we would all like to have well-written, well-documented, efficient, secure, bug-free programs
  - have we achieved this goal in computer science?
- A programming language can either encourage or discourage the introduction of errors.
  - From a software engineering perspective, we want to build secure and error-free programs.
  - this means the language needs to offer linguistic mechanisms that facilitate the production of reliable software.
    - E.g., modularity, strong typing, assertions, exceptions, high-level abstractions, security and correctness guarantees