8 Programming Paradigms
Why Study Different Paradigms?

Learn about different, more powerful programming abstractions

Higher-level description of computation

Use of abstraction
- Scalable
- Reusable
- Less error prone

Encapsulation of recursion schemes & control structures

Partial function application

Infinite data structures

Inverse computation

C
int a[10];
int s = 0;
int i;
for (i = 0; i < 10; i++) {
    s += a[i];
}
s = fold (+) 0 a

Haskell
sum [] = 0
sum (x:xs) = x + sum xs
s = sum a

Prolog
sum([],0).
sum([X|XS],S) :- sum(XS,N), S is X+N.
sum(A,S).
sum(A,9).
Granularity of Classification

The principal programming paradigms

"More is not better (or worse) than less, just different."

Concurrent

Data structures only

Nondeterminism?

Deterministic

Logic and constraint programming

Monotonic dataflow programming

Nonmonotonic dataflow programming

Concurrent logic programming

Declarative concurrent programming

Lazy functional programming

Continuation programming

Extensional programming

Descriptive declarative programming

XML, S-expression

More declarative

Named state

Unnamed state (seq. or conc.)

Less declarative


8 Programming Paradigms

What is a Programming Paradigm?

Imperative Programming

Functional Programming

Logic Programming

Object-Oriented Programming
**Paradigm**: A conceptual model underlying the theories and practice of a scientific subject (Oxford)

scientific subject = programming

**Programming Paradigm**: A conceptual model underlying the theories and practice of programming
Programming Paradigm

Programming is the process of creating programs
A program describes a particular computation

*Programming Paradigm*: A conceptual model underlying the theories and practice of computing

*Programming Paradigm*: A model of computation
Imperative Paradigm

Data are represented by a collection of state variables

Computation is a transformation of state (variables)

Formal definition of the imperative paradigm:

\[
\text{type State = [(Name,Val)]} \\
\text{type Computation = State -> State}
\]

Examples:
Turing Machine
Fortran, Pascal, C, Perl, ...
Imperative Programming Languages

Need two sublanguages:

1) Language of expressions to describe values to be stored in variables: \textbf{Expr}

2) Language of statements to describe state changes and control flow: \textbf{Stmt}

Semantics are given by two functions:

\texttt{evalE} :: \texttt{Expr} -> \texttt{State} -> \texttt{Val}

\texttt{evalS} :: \texttt{Stmt} -> \texttt{Computation}
Haskell Demo ...

Imp.hs
Functional Paradigm

*Data* are represented by *values*

*Computation* is a *function*

Formal definition of the functional paradigm:

```haskell
type Computation = Val -> Val
```

*Examples:*
- Lambda Calculus
- Lisp, Scheme, ML, Haskell, ...
Functional Programming Languages

One language of expressions to describe values and functions: Expr

Semantics are given by a function:

eval :: Expr -> Val
Haskell Demo ...

FunStatScope.hs

FunRec.hs
Logic Paradigm

Data are represented by values & relations

Computation is a relation

Formal definition of the logic paradigm:

\[
\text{type Computation} = (\text{Val}, ..., \text{Val})
\]

Examples:
Predicate Calculus
Prolog, Mercury, Curry, Twelf, ..., (SQL)

Quel, Datalog
Logic Programming Languages

One language of \textit{relations} to describe values and relations: $\text{Rel}$

Semantics are given by a function:

\[
\text{eval} :: \text{Rel} \rightarrow (\text{Val}, \ldots, \text{Val})
\]
Object-Oriented Paradigm

Data are represented by a collection of objects with state

Computation is evolution of objects (through method calls)

Formal definition of the object-oriented paradigm:

\[
\begin{align*}
\text{type State} & = [(\text{Name}, \text{Val})] \\
\text{type Methods} & = [(\text{Name}, \text{State} \rightarrow \text{State})] \\
\text{type Object} & = (\text{State}, \text{Methods}) \\
\text{type Objects} & = [\text{Object}] \\
\text{type Computation} & = \text{Objects} \rightarrow \text{Objects}
\end{align*}
\]

Examples:

Featherweight Java (& other Object Calculi)
Simula, Smalltalk, CLOS, C++, Java, C#, ...
Object-Oriented Programming Languages

Need two sublanguages for *expressions*, and *statements* (like for imperative languages), but:

The statement language needs constructs to:
(a) create objects (a group of state and methods)
(b) invoke methods (execute local state transformations)

Semantics are again given by two functions:

\[
\text{evalE} :: \text{Expr} \to \text{State} \to \text{Val} \\
\text{evalS} :: \text{Stmt} \to \text{Computation}
\]
Haskell Demo ...

Obj.hs
## Comparison of Paradigms

<table>
<thead>
<tr>
<th>In the ... paradigm,</th>
<th>computation is viewed as a ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>imperative</td>
<td>... <em>state transformation</em> that changes an input state into an output state</td>
</tr>
<tr>
<td>functional</td>
<td>... <em>function</em> that maps input to output</td>
</tr>
<tr>
<td>logic</td>
<td>... <em>relation</em> between input and output</td>
</tr>
<tr>
<td>object-oriented</td>
<td>... <em>simulation</em> through a set of interacting objects</td>
</tr>
</tbody>
</table>