8 Programming Paradigms

Why Study Different Paradigms?

```
int a[10];
int s = 0;
s += a[0];
s += a[1];
s += a[2];
s += a[3];
s += a[4];
s += a[5];
s += a[6];
s += a[7];
s += a[8];
s += a[9];
```

Higher-level description of computation

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

C

Learn about different, more powerful programming abstractions

→*Encapsulation of recursion schemes & control structures*

```
sum []     = 0
sum (x:xs) = x + sum xs

s = sum a
```

Haskell

```
s = fold (+) 0 a
```

→*Partial function application*
→*Infinite data structures*

→*Use of abstraction*
→*Scalable*
→*Reusable*
→*Less error prone*

```
sum([],0).
sum([X|XS],S) :- sum(XS,N), S is X+N.

sum(A,S).
```

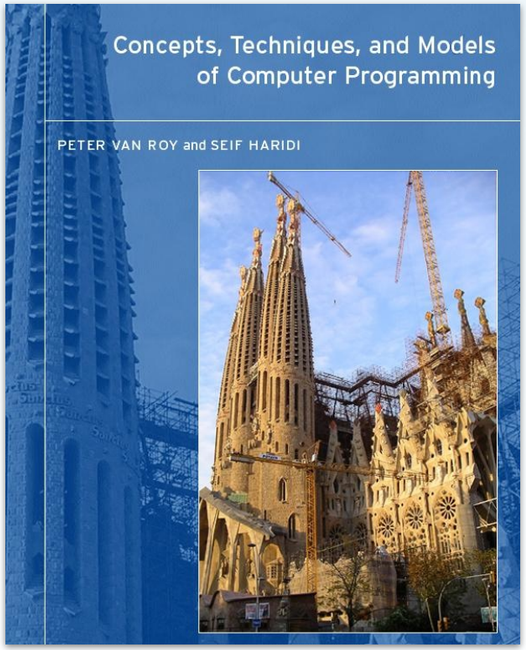
Prolog

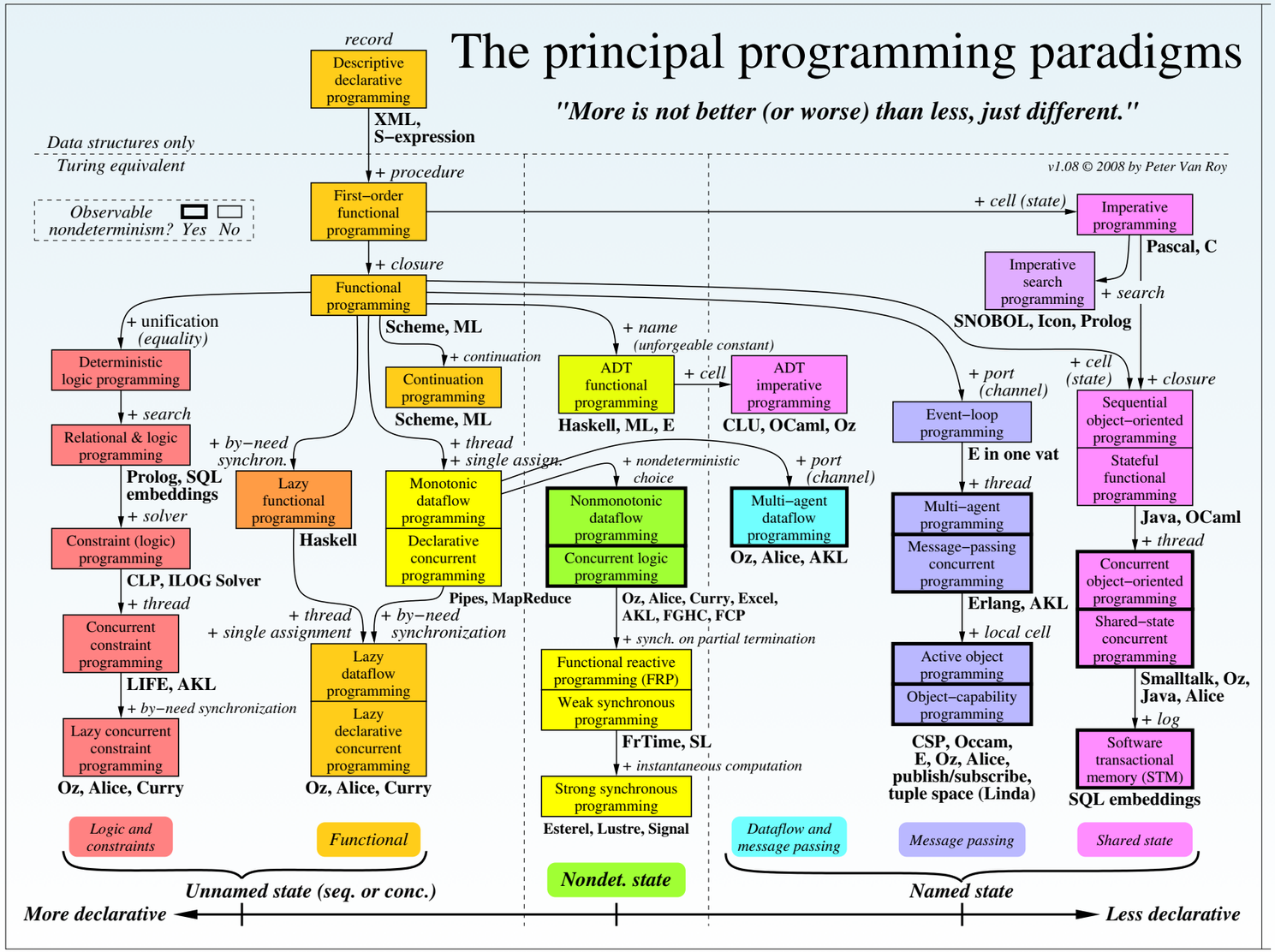
(not quite correct!)

```
sum(A,9).
```

→*Inverse computation*

Granularity of Classification





8 Programming Paradigms

What is a Programming Paradigm?

Imperative Programming

Functional Programming

Logic Programming

Object-Oriented Programming

(Programming) Paradigm

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:

```
type State = [(Name,Val)]

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: `Expr`

(2) Language of *statements* to describe state changes and control flow: `Stmt`

Semantics are given by two functions:

```
evalE :: Expr -> State -> Val

evalS :: Stmt -> Computation
```

Haskell Demo ...

Imp.hs

Functional Paradigm

Data are represented by *values*

Computation is a *function*

Formal definition of the functional paradigm:

```
type Computation = Expr -> 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:

```
type Computation = (Val, ..., Val)
```

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

Logic Programming Languages

One language of *relations* to describe values and relations: `Rel`

Semantics are given by a function:

```
eval :: Rel -> (Val, ..., 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:

```
type State   = [(Name,Val)]
type Methods = [(Name,State -> State)]
```

```
type Object      = (State,Methods)
type Objects     = [Object]
type Computation = Objects -> Objects
```

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:

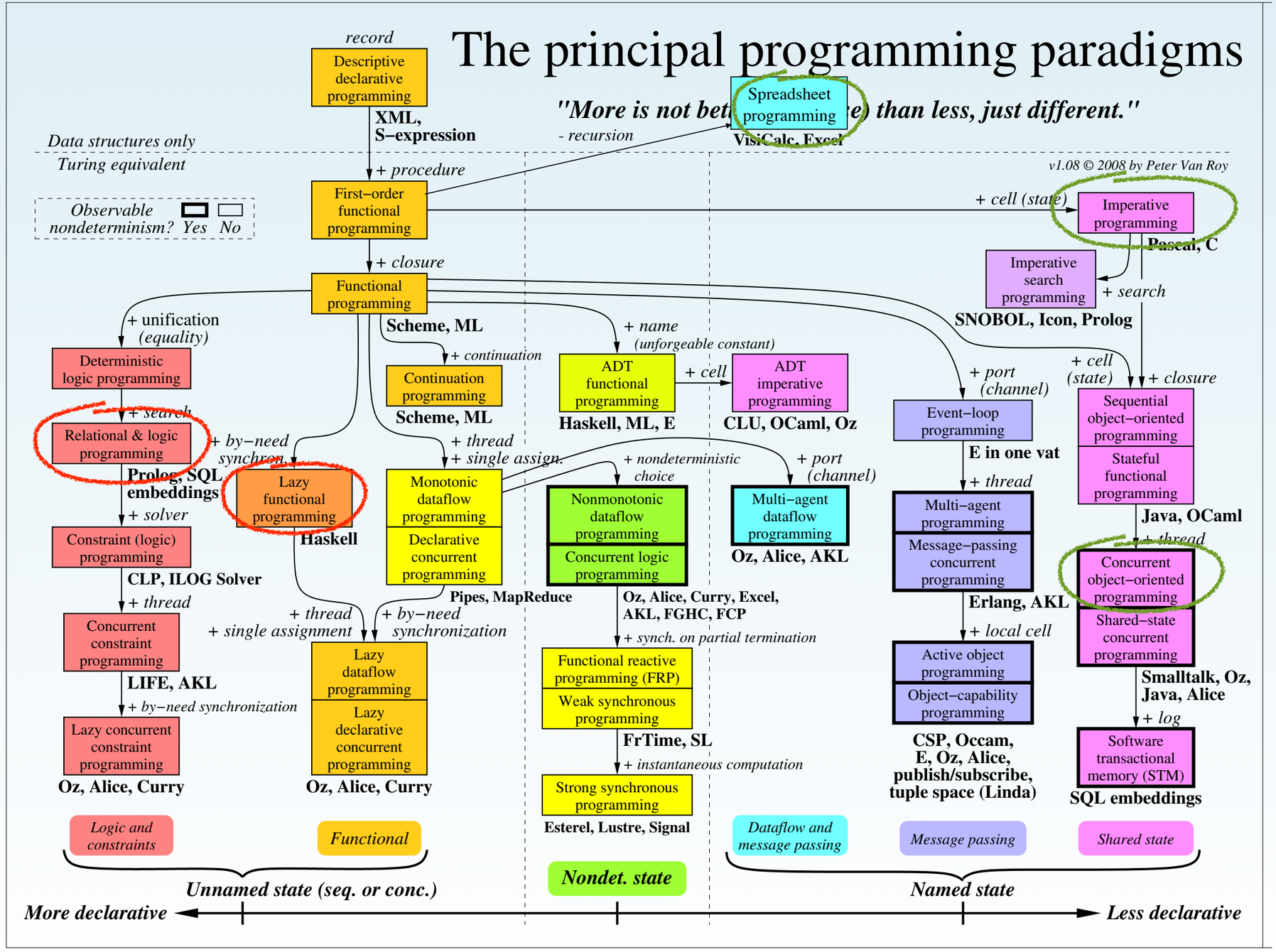
```
evalE :: Expr -> State -> Val
evalS :: Stmt -> Computation
```

Haskell Demo ...

Obj.hs

Comparison of Paradigms

In the ... paradigm,	**computation is viewed as a ...**
imperative	... *state transformation* that changes an input state into an output state
functional	... *function* that maps input to output
logic	... *relation* between input and output
object-oriented	... *simulation* through a set of interacting objects



http://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf