# Programming Paradigms

# Programming Paradigm

A way of conceptualizing what it means to perform computation and how tasks to be carried out on the computer should be structured and organized.

- Imperative : *Machine-model based*
- Functional : *Equations; Expression Evaluation*
- Logical : *First-order Logic Deduction*
- Object-Oriented : *Programming with Data Types*

# Imperative vs Non-Imperative

- *Functional/Logic programs* specify *WHAT* is to be computed abstractly, leaving the details of data organization and instruction sequencing to the interpreter.
- In constrast, *Imperative programs* describe the details of *HOW* the results are to be obtained, in terms of the underlying machine model.

# Illustrative Example

- Expression (to be computed) : a + b + c
- Recipe for Computation:
  - Intermediate Code
    - T := a + b;     T := T + c;
  - Accumulator Machine
    - Load a;  Add b;  Add c
  - Stack Machine
    - Push a;  Push b;   Add;  Push c;  Add

# Imperative vs Non-Imperative

- *Functional/Logic style* clearly separates *WHAT* aspects of a program (programmers' responsibility) from the *HOW* aspects (implementation decisions).
- An *Imperative program* contains both the specification and the implementation details, inseparably inter-twined.

# Procedural vs Functional

- Program: a sequence of instructions for a von Neumann m/c.
- Computation by instruction execution.
- Iteration.
- Modifiable or updateable variables.

- Program: a collection of function definitions (m/c independent).
- Computation by term rewriting.
- Recursion.
- *Assign-only-once* variables.

# Functional Style : Illustration

- Definition : Equations

  sum(0) = 0

  sum(n) = n + sum(n-1)

- Computation : Substituition and Replacement

  sum(2)

  = 2 + sum (2-1)

  = …

  = 3

# Paradigm vs Language

- Imperative Style

```
 i := 0;   sum := 0;
while (i < n) do
        i := i + 1;
    sum := sum + i
end;
```

  - Storage efficient

- Functional Style

```
func sum(i:int) : int;
  if i = 0
  then  0
  else i + sum(i-1)
end;
```

  - No Side-effect

# Role of Variables

- Imperative (read/write)

i 0 1 2 3 ...

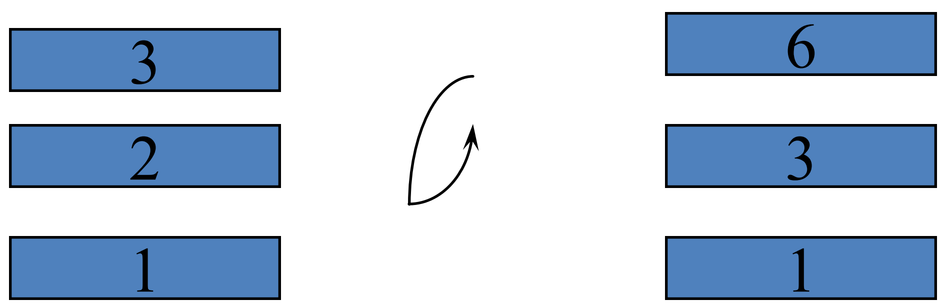
sum 0 1 3 6 ...

- Functional (read only)

i1 sum1

i2 sum2

i3 sum3



# Bridging the Gap

- Tail recursive programs can be auomatically optimized for space by translating them into equivalent while-loops.

```
func sum(i : int, r : int) : int;
    if  i = 0   then   r
    else   sum(i-1, n+r)
end
```

  – Scheme does not have loops.

# **Analogy**: Styles *vs* Formalisms

- Iteration
- Tail-Recursion
- General Recursion

- Regular Expression
- Regular Grammar
- Context-free Grammar

# Logic Programming Paradigm

- Integrates Data and Control Structures

```
edge(a,b).
edge(a,c).
edge(c,a).
path(X,X).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),  path(Z,Y).
```

# Declarative Programming

- A logic program defines a set of relations.

  This "knowledge" can be used in various ways by the interpreter to solve different queries.
- In contrast, the programs in other languages make explicit *HOW* the "declarative knowledge" is used to solve the query.

# Append in Prolog

```
append([], L, L).
append([ H | T ], L, [ H | R ]) :-
                    append(T, L, R).
```

- True statements about append relation.
  - "." and ":-" are logical connectives that stand for "*and*" and "*if*" respectively.
- Uses pattern matching.
  - "[]" and "|" stand for *empty list* and *cons* operation.

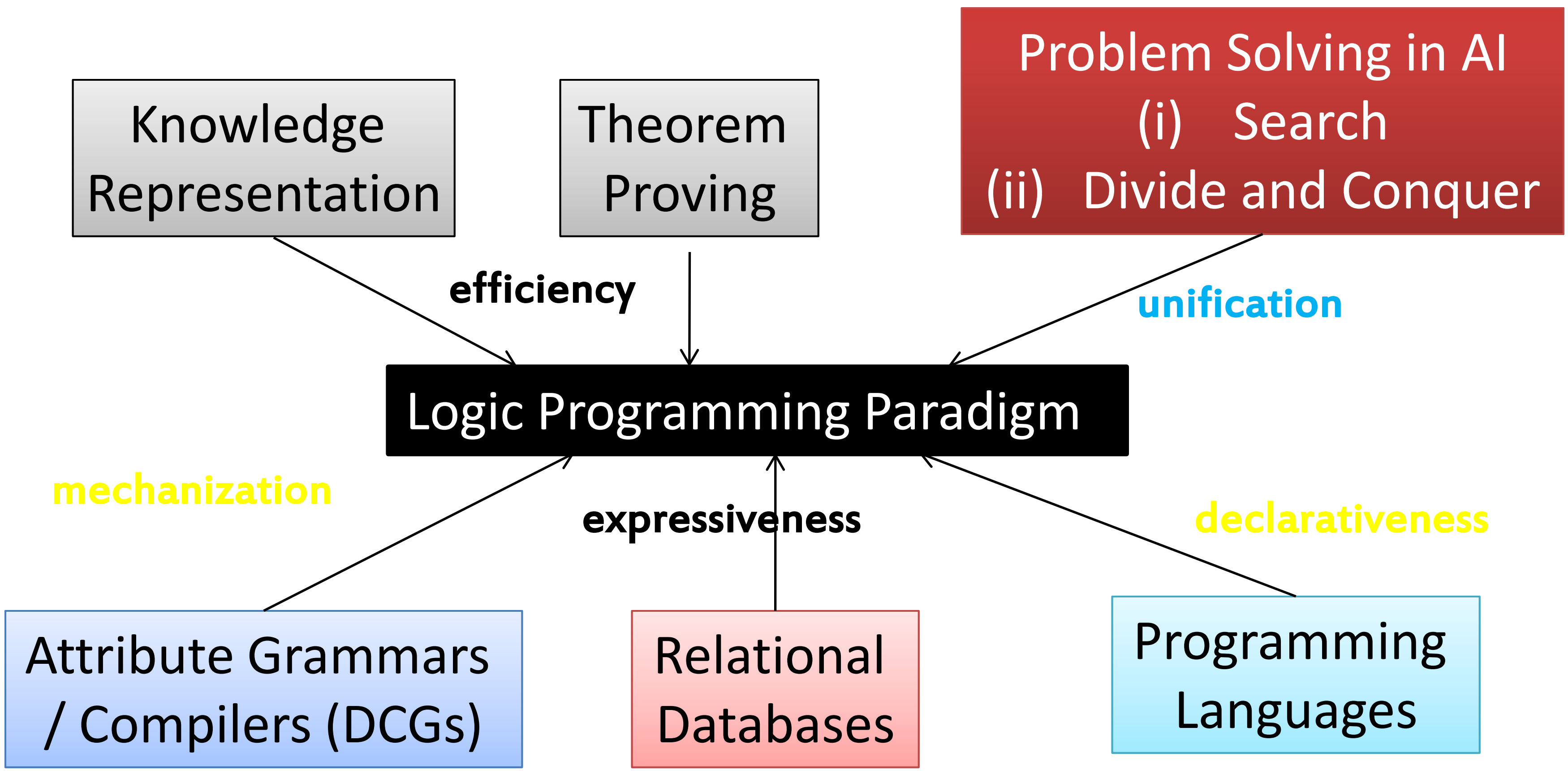
# Different Kinds of Queries

- Verification
  - sig: list x list x list
    - append([1], [2,3], [1,2,3]).
- Concatenation
  - sig: list x list -> list
    - append([1], [2,3], R).

# More Queries

- Constraint solving
  - sig: list x list -> list
    - append( R, [2,3], [1,2,3]).
  - sig: list -> list x list
    - append(A, B, [1,2,3]).
- Generation
  - sig: -> list x list x list
    - append(X, Y, Z).

# Trading expressiveness for efficiency : Executable specification



# Object-Oriented Style

- Programming with *Abstract Data Types*
  - ADTs specify/describe behaviors.
- Basic Program Unit: *Class*
  - Implementation of an ADT.
    - Abstraction enforced by encapsulation.
- Basic Run-time Unit: *Object*
  - Instance of a class.
    - Has an associated *state.*

# Procedural vs Object-Oriented

- Emphasis on procedural abstraction.
- Top-down design;

  Step-wise refinement.
- Suited for programming in the small.

- Emphasis on data abstraction.
- Bottom-up design;

  Reusable libraries.
- Suited for programming in the large.

# Integrating Heterogeneous Data

- In C, Pascal, etc., use

  Union Type / Switch Statement

  Variant Record Type / Case Statement

- In C++, Java, Eiffel, etc., use

  Abstract Classes / Virtual Functions

  Interfaces and Classes / Dynamic Binding

# Comparison : *Figures* example

- Data
  - Square
    - side
  - Circle
    - radius
- Operation (area)
  - Square
    - side * side
  - Circle
    - PI * radius * radius

- Classes
  - Square
    - side
    - area
      (= side * side)
    - Circle
      - radius
      - area
        (= PI*radius*radius)

# Adding a new operation

- Data

  ...
- Operation (area)
- Operation (perimeter)
  - Square
    - 4 * side
  - Circle
    - 2 * PI * radius

- Classes
  - Square
    - ...
    - perimeter

      (= 4 * side)
  - Circle
    - ...
    - perimeter

      (= 2 * PI * radius)

# Adding a new data representation

- Data
  - ...
  - rectangle
    - length
    - width
- Operation (area)
  - ...
  - rectangle
    - length * width

- Classes
  - ...
  - rectangle
    - length
    - width
    - area
      (= length * width)

# Procedural vs Object-Oriented

- New operations cause *additive* changes in procedural style, but require modifications to all existing “class modules” in object-oriented style.
- New data representations cause *additive* changes in object-oriented style, but require modifications to all “procedure modules”.

# Object-Oriented Concepts

- Data Abstraction (specifies behavior)
- Encapsulation (controls visibility of names)
- Polymorphism (accommodates various implementations)
- Inheritance (facilitates code reuse)
- Modularity (relates to unit of compilation)

# Example : Role of interface in decoupling

- **Client**
  - Determine the number of elements in a collection.
- **Suppliers**
  - Collections : Vector, String, List, Set, Array, etc
- **Procedual Style**
  - A client is responsible for invoking appropriate supplier function for determining the size.
- **OOP Style**
  - Suppliers are responsible for conforming to the standard interface required for exporting the size functionality to a client.

# Client in Scheme

```
(define (size C)
   (cond
     ( (vector?  C)  (vector-length C) )
     ( (pair?  C)    (length C) )
     ( (string?  C)  (string-length C) )
     (  else          “size not supported” ) )
))

(size    (vector 1 2 (+ 1 2)))
(size     ‘(one  “two”  3))
```

# Suppliers and Client in Java

```
interface  Collection  {  int size(); }
class  myVector  extends  Vector
                implements  Collection {
}
class  myString  extends  String
                implements  Collection {
   public int size() { return  length();}
}
class myArray  implements Collection {
   int[]  array;
   public int size() {return  array.length;}
}

Collection c = new  myVector();  c.size();
```