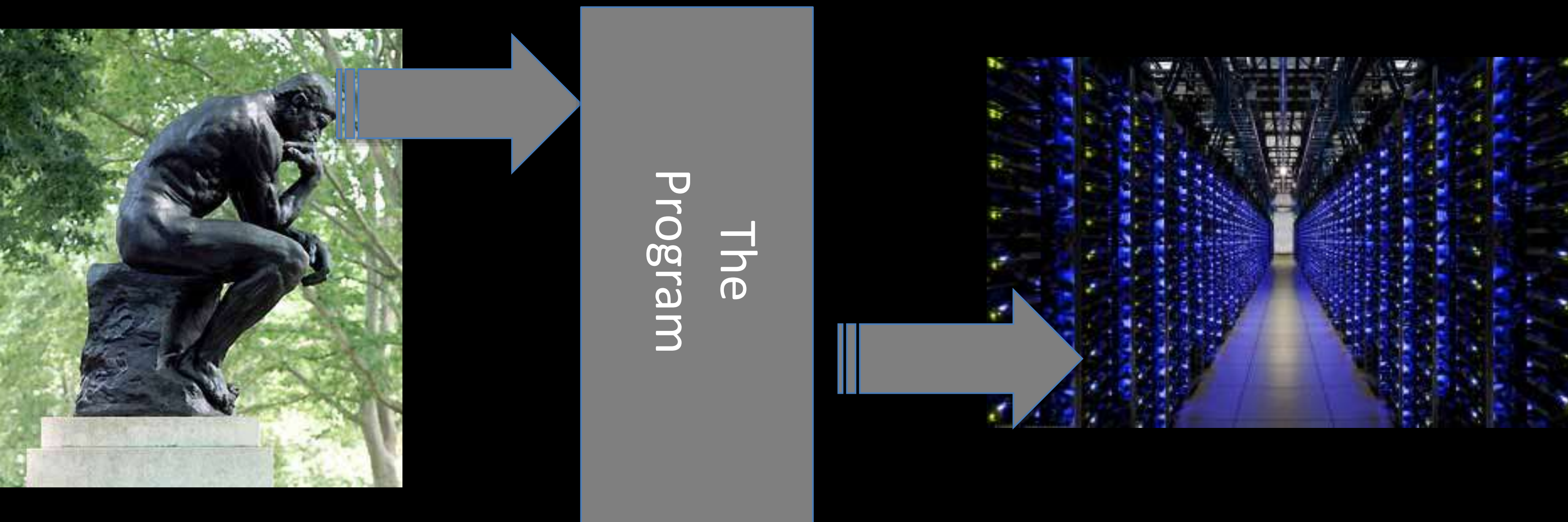
Cooperative Information Systems

Edward Blurock

PROGRAMMING PARADIGMS

Why are there so many programming languages?

SKIP



Human Computer Interface

Why are there so many programming languages?

SKIP

Human Computer Interface

The Program | The Program | The Program

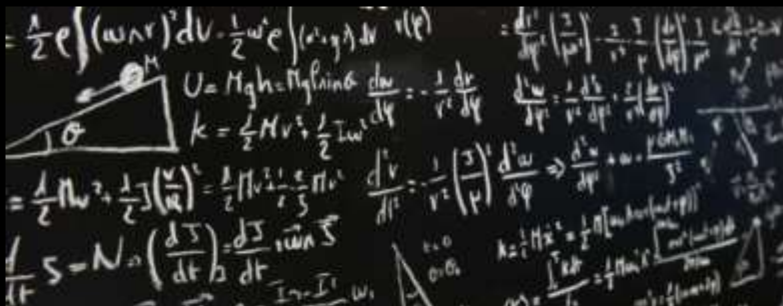
Historically, the first programs were close to the computer level

But as programming languages are evolving

Organizational structure are becoming more human

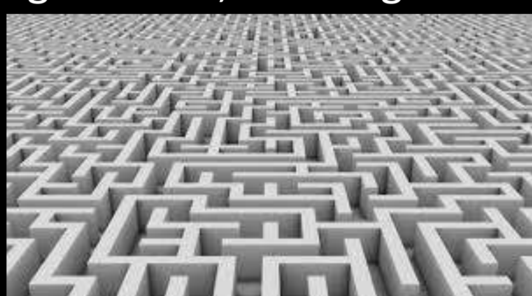
Why are there so many programming languages?

SKIP



Mathematical concepts

Organizational Concepts:
Program flow, data organization



The Program

The computer

What influences program language design?

SKIP

Mathematical Basis

FORMAL BASIS OF PROGRAMMING LANGUAGES

SKIP

Algorithm

- Abu Ja'far Muhammad ibn Musa al-Khorezmi ("from Khorezm")
  - Lived in Baghdad around 780 – 850 AD
  - Chief mathematician in Khalif Al Mamun's "House of Wisdom"
  - Author of "A Compact Introduction To Calculation Using Rules Of Completion And Reduction"

Removing negative units from the equation by adding the same quantity on the other side ("al-gabr" in Arabic)

SKIP

“Calculus of Thought”

- Gottfried Wilhelm Leibniz
  - 1646 - 1716
  - Inventor of
    calculus and binary system
  - “Calculus ratiocinator”
    human reasoning can be reduced to a formal symbolic language,
    in which all arguments would be settled by mechanical manipulation of logical concepts
  - Invented a mechanical calculator

Formalisms for Computation (1)

SKIP

- Predicate logic
  - Gottlöb Frege (1848-1925)
  - Formal basis for proof theory and automated theorem proving
  - Logic programming
    - Computation as logical deduction
- Turing machines
  - Alan Turing (1912-1954)
  - Imperative programming
    - Sequences of commands, explicit state transitions, update via assignment

Formalisms for Computation (2)

SKIP

- Lambda calculus
  - Alonzo Church (1903-1995)
  - Formal basis for all functional languages, semantics, type theory
  - Functional programming
    - Pure expression evaluation, no assignment operator
- Recursive functions & automata
  - Stephen Kleene (1909-1994)
  - Regular expressions, finite-state machines, PDAs

Programming Language

SKIP

- Formal notation for specifying computations
  - Syntax (usually specified by a context-free grammar)
  - Semantics for each syntactic construct
  - Practical implementation on a real or virtual machine
    - Translation vs. compilation vs. interpretation
      - C++ was originally translated into C by Stroustrup's Cfront
      - Java originally used a bytecode interpreter, now native code compilers are commonly used for greater efficiency
      - Lisp, Scheme and most other functional languages are interpreted by a virtual machine, but code is often precompiled to an internal executable for efficiency
    - Efficiency vs. portability

Computability

SKIP

- Function f is computable if some program P computes it
  - For any input x, the computation P(x) halts with output f(x)
  - Partial recursive functions: partial functions (int to int) that are computable

Halting Problem

Ettore Bugatti: "I make my cars to go, not to stop"

SKIP

Program Correctness

- Assert formal correctness statements about critical parts of a program and reason effectively
  - A program is intended to carry out a specific computation, but a programmer can fail to adequately address all data value ranges, input conditions, system resource constraints, memory limitations, etc.
- Language features and their interaction should be clearly specified and understandable
  - If you do not or can not clearly understand the semantics of the language, your ability to accurately predict the behavior of your program is limited

Billion-Dollar Mistake

SKIP

Failed launch of Ariane 5 rocket (1996)

- $500 million payload; $7 billion spent on development

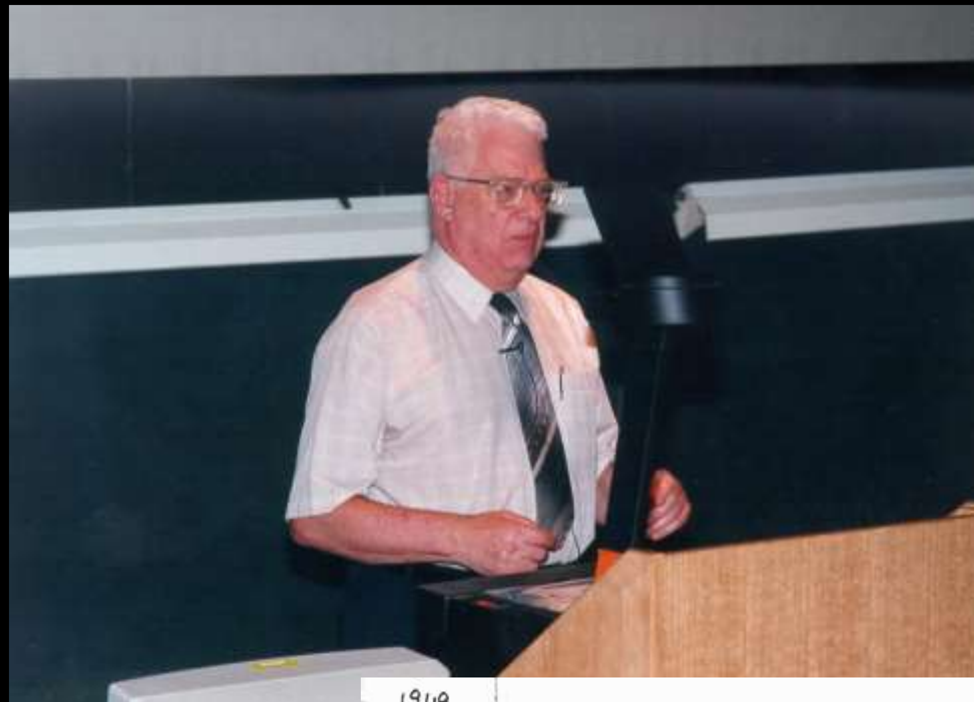
Cause: software error in inertial reference system

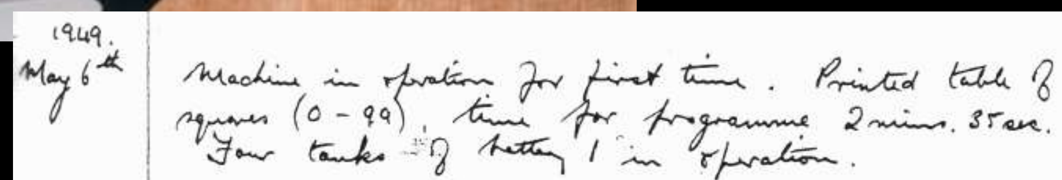
- Re-used Ariane 4 code, but flight path was different
- 64-bit floating point number related to horizontal velocity converted to 16-bit signed integer; the number was larger than 32,767; inertial guidance crashed

Evolutionary Development

HISTORY OF PROGRAMMING LANGUAGES

First modern computer – first compiler




1949.
May 6th
Machine in operation for first time. Printed table of squares (0 - 99). time for programme 2 mins. 35 sec. Four tanks of battery 1 in operation.

- David Wheeler (1927-2004)
  - University of Cambridge
  - Exceptional problem solver: hardware, software, algorithms, libraries
  - First computer science Ph.D. (1951)
  - First paper on how to write correct, reusable, and maintainable code (1951)
  - (Thesis advisor for Bjarne Stroustrup ☺)

Assembly Languages

- Invented by machine designers the early 1950s
- Mnemonics instead of binary opcodes

```
push ebp
mov ebp, esp
sub esp, 4
push edi
```

- Reusable macros and subroutines

FORTRAN

- Procedural, imperative language
  - Still used in scientific computation
- Developed at IBM in the 1950s by John Backus (1924-2007)
  - Backus's 1977 Turing award lecture (see course website) made the case for functional programming
  - On FORTRAN: "We did not know what we wanted and how to do it. It just sort of grew. The first struggle was over what the language would look like. Then how to parse expressions – it was a big problem…"
    - BNF: Backus-Naur form for defining context-free grammars

LISP

- Invented by John McCarthy (b. 1927, Turing award: 1971)
  - See original paper on course website
- Formal notation for lambda-calculus
- Pioneered many PL concepts
  - Automated memory management (garbage collection)
  - Dynamic typing
  - No distinction between code and data
- Still in use: ACL2, Scheme, …

Algol 60

- Designed in 1958-1960
- Great influence on modern languages
  - Formally specified syntax (BNF)
    - Peter Naur: 2005 Turing award
  - Lexical scoping: begin … end or {…}
  - Modular procedures, recursive procedures, variable type declarations, stack storage allocation
- "Birth of computer science" -- Dijkstra

-- Hoare

Algol 60 Sample

```
real procedure average(A,n);
   real array A; integer n;
   begin
      real sum; sum := 0;
      for i = 1 step 1 until n do
            sum := sum + A[i];
      average := sum/n
   end;
```

no array bounds

no ; here

set procedure return value by assignment

Simula 67

- Kristen Nygaard (1926-2002) and Ole-Johan Dahl (1931-2002)
  – Norwegian Computing Center
  – Oslo University
  – The start of object-oriented programming and object-oriented design

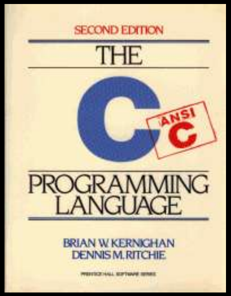
Pascal

- Designed by Niklaus Wirth
  - 1984 Turing Award
- Revised type system of Algol
  - Good data structure concepts
    - Records, variants, subranges
  - More restrictive than Algol 60/68
    - Procedure parameters cannot have procedure parameters
- Popular teaching language
- Simple one-pass compiler

C

- Bell Labs 1972 (Dennis Ritchie)
- Development closely related to UNIX
  - 1983 Turing Award to Thompson and Ritchie

- Added weak typing to B
  - int, char, their pointer types
  - Typed arrays = typed pointers
    - `int a[10]; … x = a[i];` means
      `x = *(&a[0]+i*sizeof(int))`
- Compiles to native code



C++

- Bell Labs 1979 (Bjarne Stroustrup)
  – "C with Classes" (C++ since 1983)
- Influenced by Simula
- Originally translated into C using Cfront, then native compilers
  – GNU g++
- Several PL concepts
  – Multiple inheritance
  – Templates / generics
  – Exception handling

Java

- Sun 1991-1995 (James Gosling)
  - Originally called Oak, intended for set top boxes
- Mixture of C and Modula-3
  - Unlike C++
    - No templates (generics), no multiple inheritance, no operator overloading
  - Like Modula-3 (developed at DEC SRC)
    - Explicit interfaces, single inheritance, exception handling, built-in threading model, references & automatic garbage collection (no explicit pointers!)
- "Generics" added later

Why So Many Languages?

"There will always be things we wish to say in our programs that in all languages can only be said poorly."

- Alan Perlis

What's Driving Their Evolution?

- Constant search for better ways to build software tools for solving computational problems
  - Many PLs are general purpose tools
  - Others are targeted at specific kinds of problems
    - For example, massively parallel computations or graphics
- Useful ideas evolve into language designs
  - Algol → Simula → Smalltalk → C with Classes → C++
- Often design is driven by expediency
  - Scripting languages: Perl, Tcl, Python, PHP, etc.
    - "PHP is a minor evil perpetrated by incompetent amateurs, whereas Perl is a great and insidious evil, perpetrated by skilled but perverted professionals."   - Jon Ribbens

What Do They Have in Common?

- Lexical structure and analysis
  – Tokens: keywords, operators, symbols, variables
  – Regular expressions and finite automata
- Syntactic structure and analysis
  – Parsing, context-free grammars
- Pragmatic issues
  – Scoping, block structure, local variables
  – Procedures, parameter passing, iteration, recursion
  – Type checking, data structures
- Semantics
  – What do programs mean and are they correct

Core Features vs. Syntactic Sugar

- What is the core high-level language syntax required to emulate a universal Turing machine?
  - What is the core syntax of C?
    - Are ++, --, +=, -=, ?:, for/do/while part of the core?
- Convenience features?
  - Structures/records, arrays, loops, case/switch?
  - Preprocessor macros (textual substitution)
  - Run-time libraries
    - String handling, I/O, system calls, threads, networking, etc.
  - "Syntactic sugar causes cancer of the semicolons"

    \- Alan Perlis

SKIP

Where Do Paradigms Come From?

- Paradigms emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas
  - Thomas Kuhn. "The Structure of Scientific Revolutions."
- Programming paradigms are the result of people's ideas about how programs should be constructed
  - … and formal linguistic mechanisms for expressing them
  - … and software engineering principles and practices for using the resulting programming language to solve problems

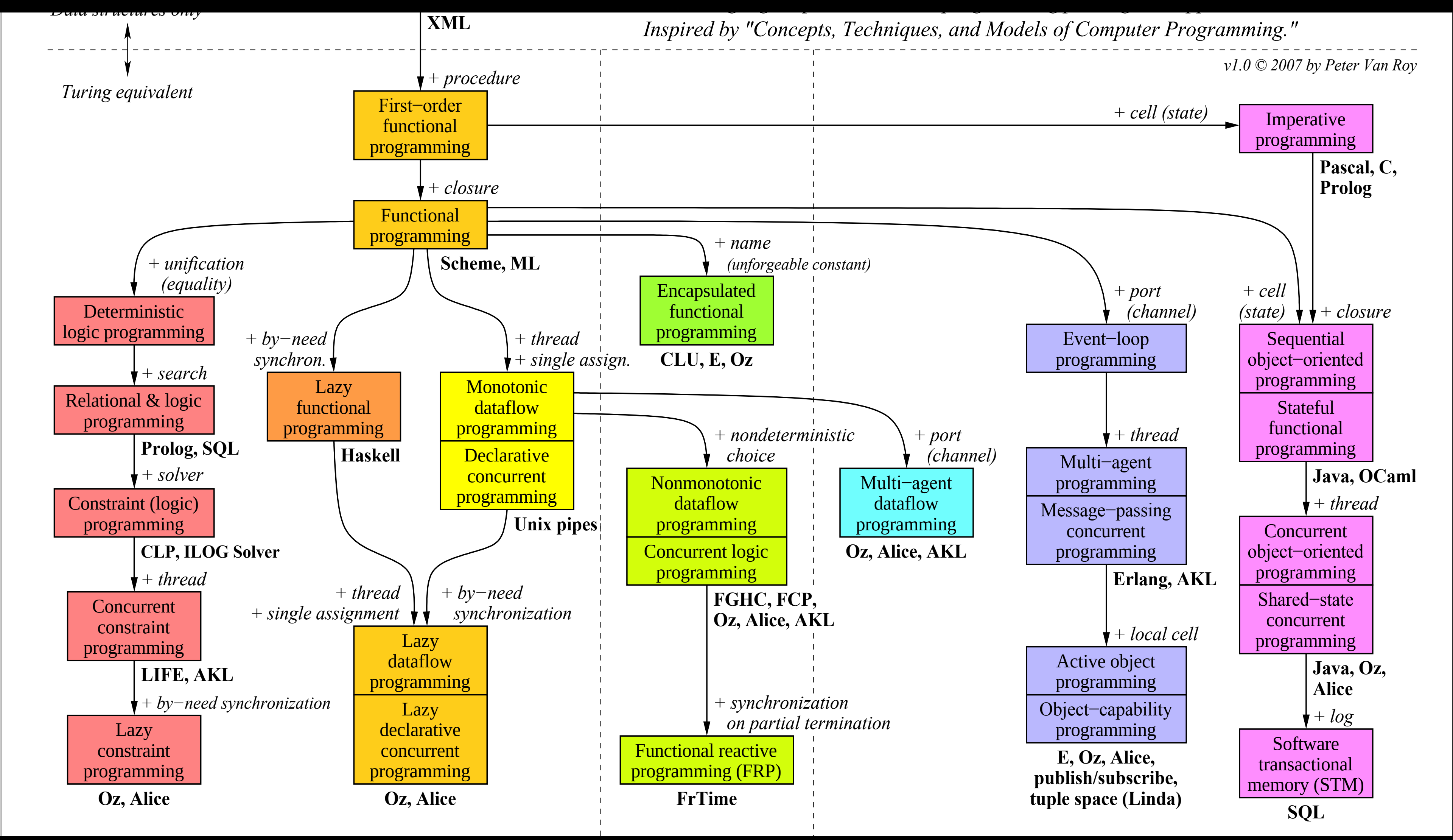
Paradigms

- Programming paradigm is a fundamental style of computer programming

- Paradigms differ in concepts and abstractions used to represent the elements of program

Paradigms

SKIP



Major Paradigms

- Imperative
- Functional
- Declarative
- Object Oriented
- Event-Driven
- Stochastic

There are many ways to categorize programming paradigms.

It should also be said that they are not mutually exclusive.

Programming Paradigms

IMPERATIVE LANGUAGES

Imperative Programming

- Derived from latin word *imperare* means “to command”
- It is based on commands that update variables in storage
- Is a programming paradigm that describes computation in terms of statements that change a program state.

SKIP

Contd..

- It defines sequences of commands for the computer to perform
- Imperative programming is characterized by programming with a state and commands

SKIP

Contd..

- In imperative programming, a name may be assigned to a value and later reassigned to another value.
- A name is tied to two bindings, a binding to a location and to a value.
- The location is called the l-value and the value is called the r-value.

SKIP

Contd..

- For example,
  - X := X+2
- Assignment changes the value at a location.
- A program execution generates a sequence of states

SKIP

Unstructured Commands

- The unstructured commands contains:
  - assignment command,
  - sequential composition of commands,
  - a provision to identify a command with a label,
  - unconditional and conditional GOTO commands

SKIP

L-value and r-value

- l-value: address/location
  - lifetime
  - memory allocation
- r-value: contents/encoded value
  - initialization
  - constants

* binding

SKIP

Iteration

- For statement
  - Loop control variable
- The while loop
- while vs do-while
- do-while versus repeat-until
- Iteration using goto

Goto statement

SKIP

Loops a GOTO (or similar) statement

The GOTO jumps to a specified location (label or address)

The index is incremented until the end is reached

an index involved

```
     i=1
     factorial = 1;
loop:
     factorial  = factorial * I
     if( i=n) goto exit
     goto loop
exit
```

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the

SKIP

Structured Programming

- The goal of structured programming is to provide control structures that make it easier to reason about imperative programs.

SKIP

Contd..

- an IF statement corresponds to an If condition then command and a DO statement corresponds to a While condition Do command.

  - IF *guard* --> *command* FI=if *guard* then *command*
  - DO *guard* --> *command* OD=while *guard* do *command*

Iteration

SKIP

Repetition of a block of code

The index is incremented until the end is reached

Once again involves a iteration counter

an index involved

```
 i=1
 factorial = 1;
while( i <= n) {
    factorial  = factorial * i
    i = i + 1
    }
```

```
factorial = 1;
for  i=1 to n    {
        factorial  = factorial * I
        }
```

SKIP

Programming Paradigms: Functional

- A program in this paradigm consists of *functions* and uses functions in a similar way as used in mathematics
  - Program execution involves functions calling each other and returning results. There are no variables in functional languages.
- Example functional languages include: ML, Miranda™, Haskell
- Advantages
  - Small and clean syntax
  - Better support for reasoning about programs
  - They allow functions to be treated as any other data values.
  - They support programming at a relatively higher level than the imperative languages
- Disadvantages
  - Difficulty of doing input-output
  - Functional languages use more storage space than their imperative cousins

SKIP

Possible Benefits

- The ability to re-use the same code at different places in the program without copying it.
- An easier way to keep track of program flow than a collection of "GOTO" or "JUMP" statements

Functional Programming

- It treats computation as the evaluation of mathematical functions and avoids state and mutable data.

- It emphasizes the application of functions, in contrast to the imperative programming style

Recursion

Numerische Mathematik 2, 312--318 (**1960**)

Recursive Programming*

By

E. W. DIJKSTRA

The Aim

If every subroutine has its own private fixed working spaces, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need *simultaneously*, and the available memory space is therefore used rather uneconomically. Furthermore—and this is a more serious objection—it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on.

Content of Recursion

- **Base case(s).**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.
- ***Recursive calls.***
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

```
factorial(n)  {
      if(n=1) return 1
     return factorial(n-1)*n
}
```

SKIP

How do I write a recursive function?

- Determine the size factor
  - The number: smaller number, smaller size
- Determine the base case(s)
  - The case for n=1, the answer is 1
- Determine the general case(s)
  - The recursive call: factorial(n)=factorial(n-1)*n
- Verify the algorithm

  (use the "Three-Question-Method")

```
factorial(n) {
    if(n=1) return 1
    return factorial(n-1)*n
}
```

SKIP

Three-Question Verification Method

The Base-Case Question:

Is there a nonrecursive way out of the function, and does the routine work correctly for this "base" case?

The Smaller-Caller Question:

Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

The General-Case Question:

Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Stacks
in recursion

```
factorial(n)
    If (n=1)
        return 1
    else
        return factorial(n-1)
```

$n! = n*(n-1)*(n-2)*(n-3)*......*\ 1$

Factorial(1) return 1

Factorial(2) Return 2

Factorial(3) Return 6

Factorial(4) Return 24

Factorial(5) Return 120

$5! = 5*4*3*2*1$

Deep recursion can result in running out of memory

$5!=120$

Programming Paradigms

DECLARATIVE PROGRAMMING

Declarative Programming

Not a recipe

Does not explicitly list command or steps
that need to be carried out to achieve the results.

Set of Declarations

The domain is described through a set of declarations

An external "engine" interprets these declarations to perform a task

separates the process of
stating a problem from the *process of solving* the problem.

SKIP

Programmed Behavior

Digger Wasps are predators that can sting and paralyze prey insects.
They construct a protected “nest
and then stock it with captured insects.
The wasps lay their eggs in the provisioned nest.
When the wasp larvae hatch, they feed on the paralyzed insects.

1. **Dig nest**
2. **Place paralyzed insect at nest entrance**
3. **Go in and inspect nest**
4. **Come out**
5. **Take paralyzed insect in nest**

SKIP

Programmed Behavior

1. Dig nest
2. Place paralyzed insect at nest entrance
3. Go in and inspect nest
4. Come out
5. Take paralyzed insect in nest

If during step 3, when the insect is inside the nest, the researcher removes the insect
The insect will go back to step 2, retrieve the insect and, again step 3, inspect the nest.

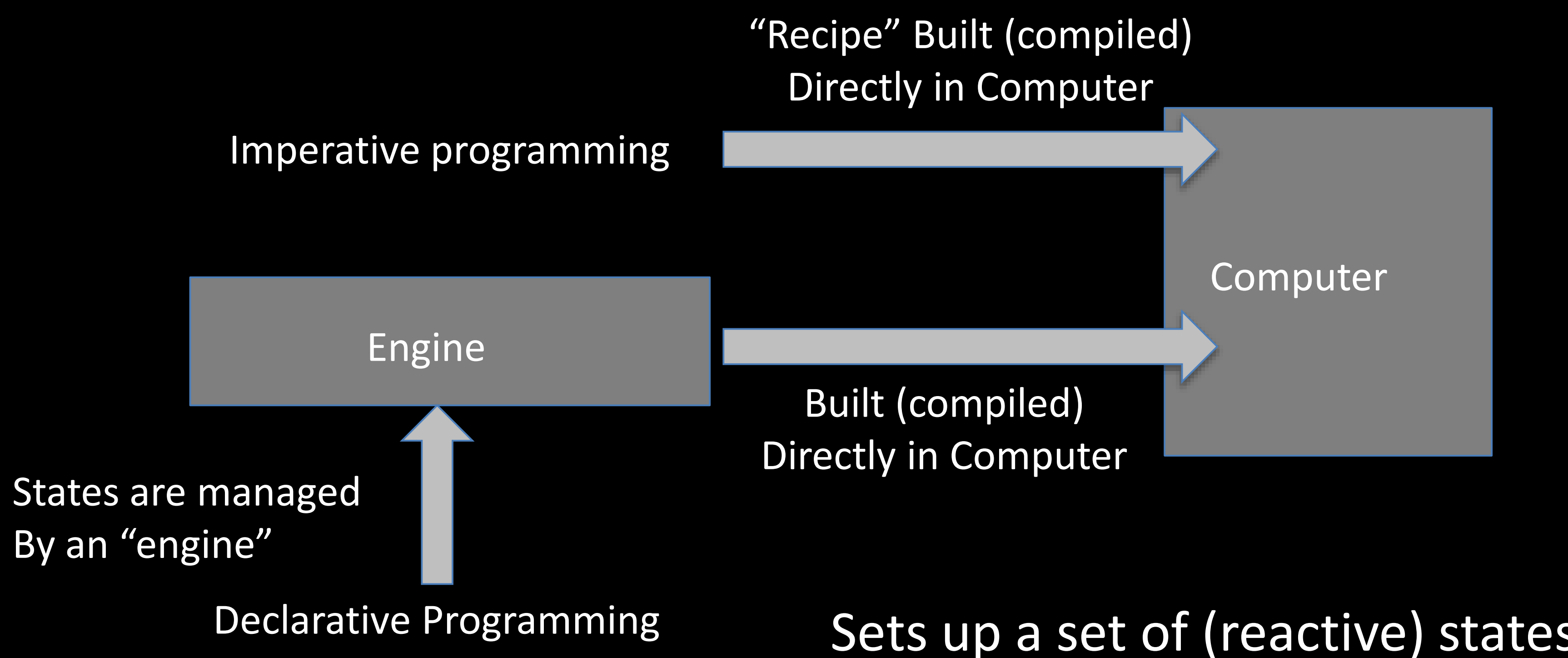
Programmed Behavior

1. Dig nest
2. Place paralyzed insect at nest entrance
3. Go in and inspect nest
4. Come out
5. Take paralyzed insect in nest

Imperative View:
A recipe, step 1, before step 2, before step 3….
Fixed programming can lead to (unexpected) problems.

Declarative Paradigm

Not a recipe for the computer



Declarative programming

Expresses the **logic of a computation** without describing its **control flow**.

```
factorial(1,1)
factorial(N,F) :-
        N1 is N-1,
        factorial(N1,F1),F is N*F1.
```

The problem is (mathematically) described
The engine takes care of the solving/using the description

Solving predicate logic is the engine

Declarative Programming

PROLOG:

$(p \wedge q \wedge \ldots \wedge t) \rightarrow u$

Horn clause predicate:
P(X,Y,…) :=
Condition1(…),
Condition2(…),
....

If these conditions are true,
then the predicate is true

Backward chaining:
examine conditions to determine truth of predicate

Rules

- Consider the following sentence :
  - 'All men are mortal'
- We can express this as :
  - mortal(X) :- human(X).

- Let us define the *fact* that Socrate is a human.
  - mortal(X) :- human(X).

    human(socrate).

Contd..

- Now if we ask to prolog :
  - ?- mortal(socrate).

- What prolog will respond ?

- Why ?

SKIP

Constraint Logic Programming

```
factorial(1,1)        factorial(N,F) :-
                          N1 is N-1,
                          factorial(N1,F1),F is N*F1.
```

Factorial(5,F) — **Returns F=120**

Factorial(N.120) — **Creates an instantiation error**

PROLOG has no knowledge of Real or Integer numbers
Mathematical manipulations cannot be made

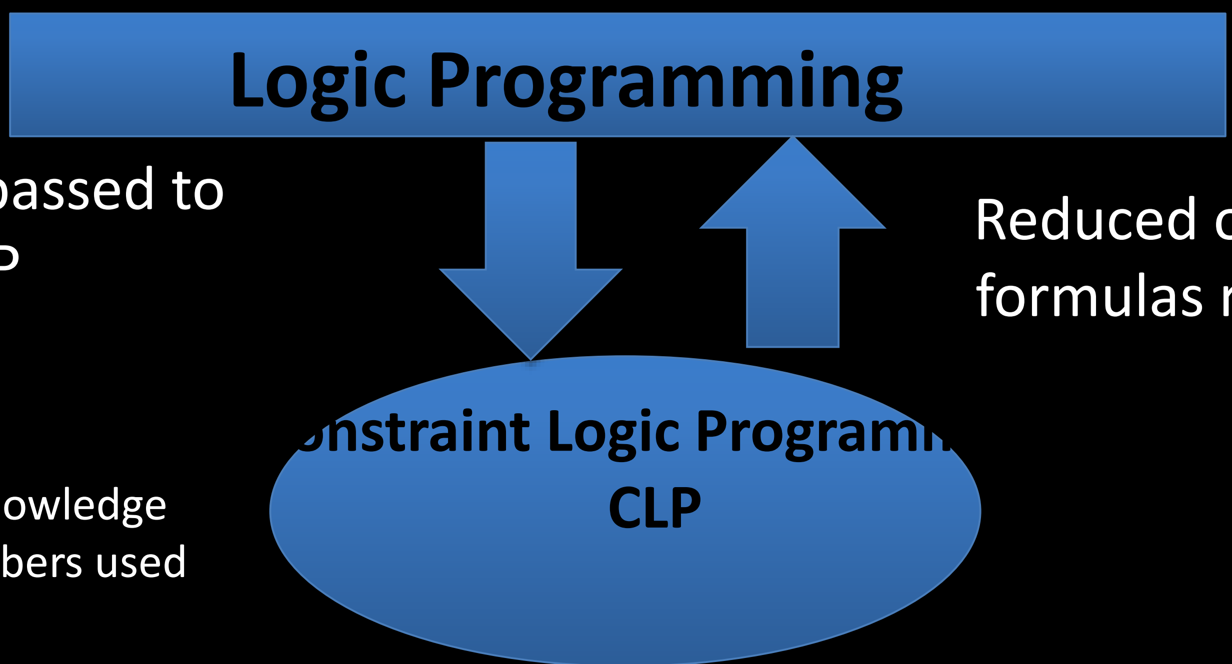
SKIP

Constraint Logic Programming

factorial(1,1)

```
factorial(N,F) :-
        N1 is N-1,
        factorial(N1,F1),F is N*F1.
```

Formulas passed to CLP



Reduced or solved formulas returned

Mathmatical knowledge about the numbers used

SKIP

Declarative Programming

Subclass:
Reactive systems

Response to a set of conditions

Declaration:
1. The set of conditions
2. The response

Declarative Programming

Expert Systems (is the engine):

A set of rules.

The rule executes if the conditions hold.

Rules:

1. If X croaks and eats flies - Then X is a frog
2. If X chirps and sings - Then X is a canary
3. If X is a frog - Then X is green
4. If X is a caary - Then X is yellow

Facts

1. Fritz croaks
2. Fritz eats flies

1. Fritz croaks and Fritz eats flies
   Based on logic, the computer can derive:
2. Fritz croaks and eats flies
   Based on rule 1, the computer can derive:
3. Fritz is a frog
   Based on rule 3, the computer can derive:
4. Fritz is green.

SKIP

Declarative Programming

GUI Interface:

An action, activated by conditions, produces a response.

Action **Response**

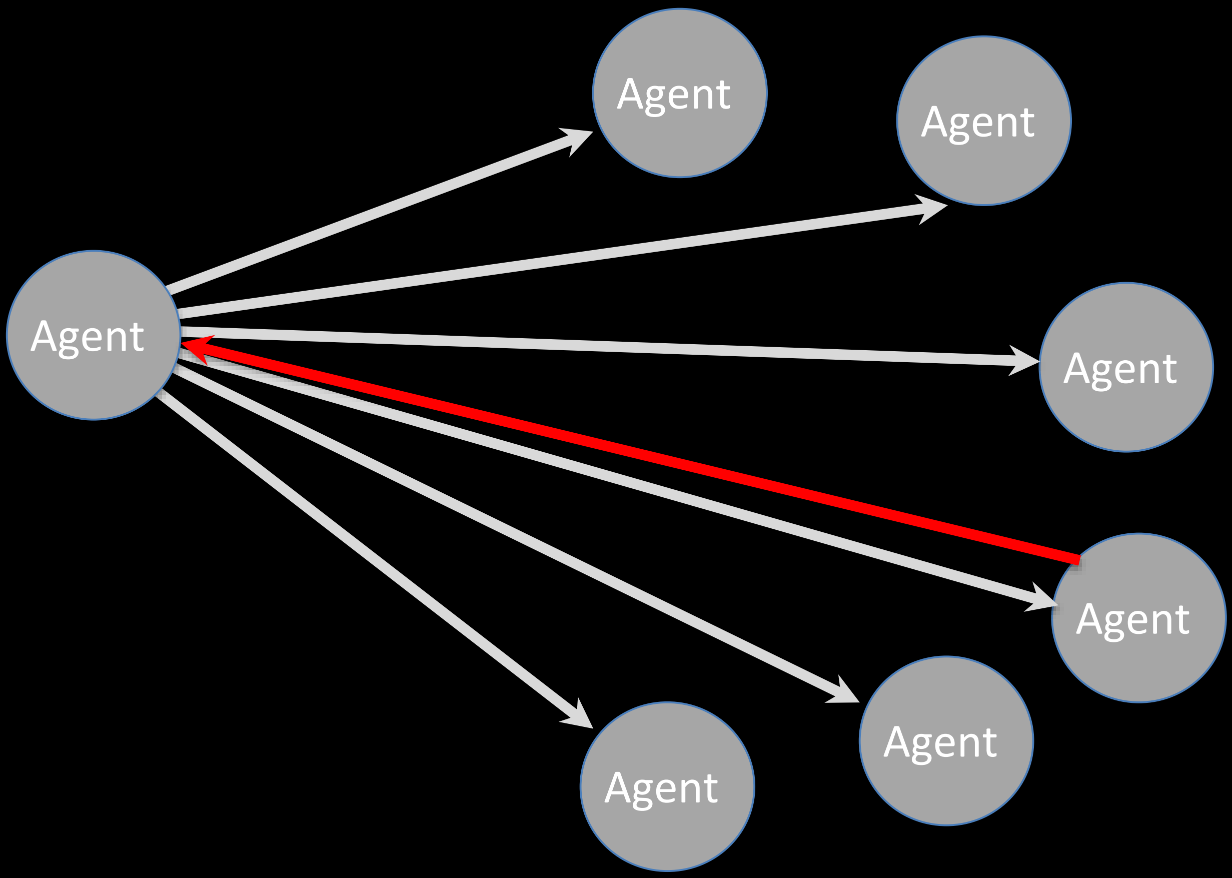
The
System
Explodes!

SKIP

Declarative Programming

System of agents (on the internet):

An agent responds, if it sees the right conditions.



Declarative Programming

Environment	Declaration	Engine
PROLOG	Horn Clause	Predicate Logic: Backward chaining
Expert System	A rule or fact	Expert System: Forward chaining
GUI	An action function	GUI/Operating system
System of Agents	Program within agent	Internet message passing

SKIP

Contd..

- It is a paradigm where we focus real life objects while programming any solution.

- We actually write behaviours of our programming objects, those behaviours are called methods in objected oriented programming.

SKIP

Principal advantage

- They enable programmers to create modules that do not need to be changed when a new type of object is added.

- A programmer can simply create a new object that inherits many of its features from existing objects.

Fundamental Concepts

- Class
- Object
- Instance
- Method
- Message passing
- Inheritance
- Abstraction
- Encapsulation
- Polymorphism
- Decoupling

Main features

- Encapsulation:
  - a logical boundary around methods and properties
- Inheritance
- Re-usability
  - method overloading and overriding
- Information Hiding
  - is achieved through "Access Modifiers"

Probabilistic Algorithms

SKIP

Non-deterministic

No exact control program flow

Leaves Some of Its Decisions To Chance

Outcome of the program in different runs is not necessarily the same

Monte Carlo Methods

Always Gives an answer
But not necessarily Correct
The probability of correctness go es up with time

Las Vegas Methods

Never returns an incorrect answer
But sometimes it doesn’t give an answer

Probabilistic Algorithms

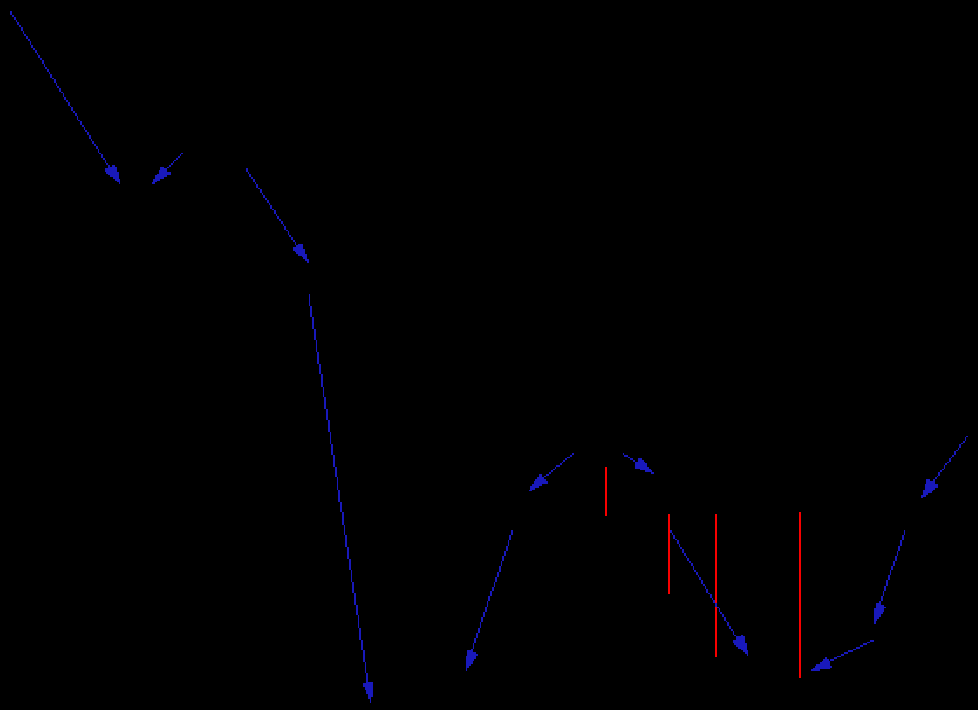
SKIP

Closer to human reasoning and problem solving

(for hard problems we don't follow strict deterministic algorithms)

Probabilistic Algorithms in optimization:

Finding
local and global
minimum

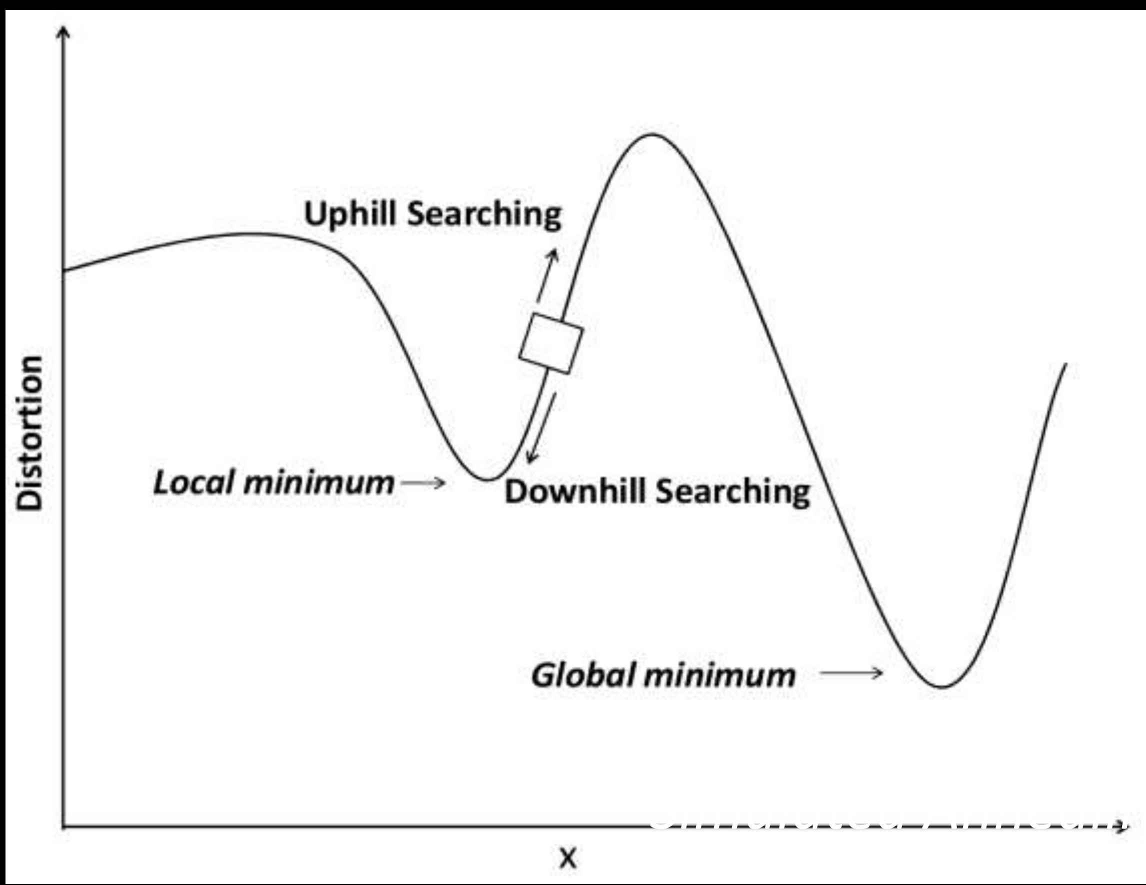


Probabilistic Algorithms

Classic gradient optimization find local minimum

The search path is **always downhill** toward minimum

Probabilistic algorithms allow search to go **uphill sometimes**



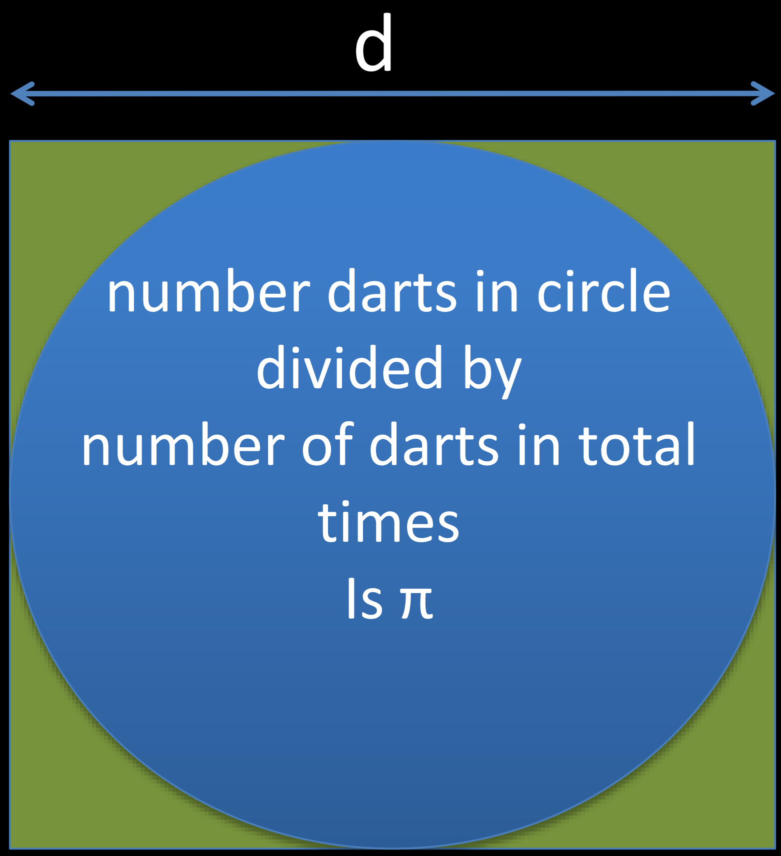
tep

g to find global minimum

Probabilistic Algorithms

SKIP

Calculate pi with a dart board



Area of square

d^2

Area of Circle:

Probability dart will be in circle

Monte Carlo Method

Always Gives an answer
But not necessarily Correct
The probability of correctness goes up with time

SKIP

Which Programming Paradigm is Best?

- Which of these paradigms is the best?
- The most accurate answer is that there is no best paradigm.
- No single paradigm will fit all problems well.
- Human beings use a combination of the models represented by these paradigms.
- Languages with features from different paradigms are often too complex.
- So, the search of the ultimate programming language continues!

Design Choices

- C: Efficient imperative programming with static types
- C++: Object-oriented programming with static types and ad hoc, subtype and parametric polymorphism
- Java: Imperative, object-oriented, and concurrent programming with static types and garbage collection
- Scheme: Lexically scoped, applicative-style recursive programming with dynamic types
- Standard ML: Practical functional programming with strict (eager) evaluation and polymorphic type inference
- Haskell: Pure functional programming with non-strict (lazy) evaluation.

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 updatable variables..

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

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.