CMPUT 325 - Language Paradigms

Dr. B. Price & Dr. R. Greiner

7th September 2004

Programming Paradigms

- Real languages draw upon multiple paradigms

Programming Paradigms

- Real languages draw upon multiple paradigms
- We consider pure programming paradigms

Programming Paradigms

- Real languages draw upon multiple paradigms
- We consider pure programming paradigms
- First, we survey the major paradigms

Programming Paradigms

- Real languages draw upon multiple paradigms
- We consider pure programming paradigms
- First, we survey the major paradigms
- Then, we examine a subset of paradigms in detail

The Procedural Paradigm

- First computer languages were *procedural* (assembly, Fortran, etc.)

The Procedural Paradigm

- First computer languages were *procedural* (assembly, Fortran, etc.)
- Emphasized in introductory courses and

The Procedural Paradigm

- First computer languages were *procedural* (assembly, Fortran, etc.)
- Emphasized in introductory courses and
- Form basis of the majority of real-world programming

The Procedural Paradigm

- First computer languages were *procedural* (assembly, Fortran, etc.)
- Emphasized in introductory courses and
- Form basis of the majority of real-world programming
- The key concept: **altering a value**
  - altering variables by assignment
  - altering variables by transformation (applying multiplication)
  - altering environments (procedure call)
  - altering I/O (assign values to outputs, assigning vars to inputs)

The Procedural Paradigm

- First computer languages were *procedural* (assembly, Fortran, etc.)
- Emphasized in introductory courses and
- Form basis of the majority of real-world programming
- The key concept: **altering a value**
  - altering variables by assignment
  - altering variables by transformation (applying multiplication)
  - altering environments (procedure call)
  - altering I/O (assign values to outputs, assigning vars to inputs)
- a.k.a imperative: you tell the program which (altering) actions to take

Procedural Sorting

- Sort an array of elements set T procedurally:

```
void naive_bubble_sort(int *T, int n) {



                          }
```

Procedural Sorting

- Sort an array of elements set T procedurally:

```
void naive_bubble_sort(int *T, int n) {
   for(int i=0; i< n; i++)
      for(int j=0; j<n-1; j++)

                                  }
```

- We loop by repeatedly altering indicies

Procedural Sorting

- Sort an array of elements set T procedurally:

```
void naive_bubble_sort(int *T, int n) {
   for(int i=0; i< n; i++)
      for(int j=0; j<n-1; j++)
         if( T[j] < T[j+1]) {
            int tmp = T[j];
            T[j] = T[j+1];
            T[j+1] = tmp;    } }
```

- We loop by repeatedly altering indicies
- We sort by pair-wise altering elements that are out of order

Procedural Sorting

- Sort an array of elements set T procedurally:

```
void naive_bubble_sort(int *T, int n) {
   for(int i=0; i< n; i++)
      for(int j=0; j<n-1; j++)
         if( T[j] < T[j+1]) {
            int tmp = T[j];
            T[j] = T[j+1];
            T[j+1] = tmp;    } }
```

- We loop by repeatedly altering indicies
- We sort by pair-wise altering elements that are out of order
- Original array is altered to contain new elements

Comments on Procedural Languages

- New computations destroy results of old computations

Comments on Procedural Languages

- New computations destroy results of old computations
- Procedure1 can inadvertently modify data that violates the assumptions of Procedure2

Comments on Procedural Languages

- New computations destroy results of old computations
- Procedure1 can inadvertently modify data that violates the assumptions of Procedure2
- Dominant computational metaphors are:
  - Sequence (statements in list)
  - Conditional (`if then else`)
  - Iteration (`for, do, while`)

Comments on Procedural Languages

- New computations destroy results of old computations
- Procedure1 can inadvertently modify data that violates the assumptions of Procedure2
- Dominant computational metaphors are:
  - Sequence (statements in list)
  - Conditional (`if then else`)
  - Iteration (`for, do, while`)
- Key to understanding a pure procedural progam:
  "How does program alter the data?"

Commonly Associated Features

Typically but not necessarily:

Commonly Associated Features

Typically but not necessarily:

- User is responsible for allocating space for variables

Commonly Associated Features

Typically but not necessarily:

- User is responsible for allocating space for variables
- Space is often rigidly typed - it can only be used for one type of data

Examples of Procedural Languages

How many can you name?

Examples of Procedural Languages

How many can you name?

- Assembly Languages: used to implement low-level drivers & interfaces
- Mainstream languages:
  - Fortran (used in sciences)
  - C (general & systems programming)
  - ADA (used in military and research)
  - PERL, Basic & Javascript (used in scripting and interfaces)
  - APL, S, M: highly specialized languages for mathematics
  - LOGO: used in children's education
- Scripting languages: csh, bash, tcl, etc.
- Other languages: Pascal, COBOL, PL/I, Algol

Object-Oriented Paradigm

- Extension of procedural paradigm

Object-Oriented Paradigm

- Extension of procedural paradigm
- Emphasis is **objects** and their relationships (not processes).

Object-Oriented Paradigm

- Extension of procedural paradigm
- Emphasis is **objects** and their relationships (not processes).
- Encapsulates procedures and associated data into unit

Object-Oriented Paradigm

- Extension of procedural paradigm
- Emphasis is **objects** and their relationships (not processes).
- Encapsulates procedures and associated data into unit
  - allows guarantees of invariant properties of the unit

Object-Oriented Sorting

- New class: `SortedSet`

Object-Oriented Sorting

- New class: `SortedSet`
- Data and operations of SortedSet's are defined together
  - Inserting and removing elements, importing sets, etc. presever sortedness property

Object-Oriented Sorting

- New class: `SortedSet`
- Data and operations of SortedSet's are defined together
  - Inserting and removing elements, importing sets, etc. presever sortedness property
- To sort elements, we simply insert the elements of T into the SortedSet

```
SortedSet S = new SortedSet();
S.import(T);
int max = S.first()
```

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms
- Procedural: sorted array can become unsorted

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms
- Procedural: sorted array can become unsorted
  - Change value of element in array

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms
- Procedural: sorted array can become unsorted
  - Change value of element in array
  - Not possible on a sorted set

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms
- Procedural: sorted array can become unsorted
  - Change value of element in array
  - Not possible on a sorted set
- Objects control how data is altered

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms
- Procedural: sorted array can become unsorted
  - Change value of element in array
  - Not possible on a sorted set
- Objects control how data is altered
- Encapsulation can improve maintainability and verifiability

Comments on Object-Oriented Approach I

- Underlying implementation will typically be expressed in procedural terms
- Procedural: sorted array can become unsorted
  - Change value of element in array
  - Not possible on a sorted set
- Objects control how data is altered
- Encapsulation can improve maintainability and verifiability
- Encapsulation can be broken by derived subclasses

Comments on Object-Oriented Approach II

- Difficult issues: multiple inheritance

Comments on Object-Oriented Approach II

- Difficult issues: multiple inheritance
- Typically but not necessarily object-oriented languages have:

Comments on Object-Oriented Approach II

- Difficult issues: multiple inheritance
- Typically but not necessarily object-oriented languages have:
  - Garbage collection: language allocates and deallocates variables as necessary

Comments on Object-Oriented Approach II

- Difficult issues: multiple inheritance
- Typically but not necessarily object-oriented languages have:
  - Garbage collection: language allocates and deallocates variables as necessary
  - Free typing: parameters and variables are not statically typed

Comments on Object-Oriented Approach II

- Difficult issues: multiple inheritance
- Typically but not necessarily object-oriented languages have:
  - Garbage collection: language allocates and deallocates variables as necessary
  - Free typing: parameters and variables are not statically typed
  - Polymorphism: the same procedure (method) can be applied to various data types

Comments on Object-Oriented Approach II

- Difficult issues: multiple inheritance
- Typically but not necessarily object-oriented languages have:
  - Garbage collection: language allocates and deallocates variables as necessary
  - Free typing: parameters and variables are not statically typed
  - Polymorphism: the same procedure (method) can be applied to various data types
- Inconsistency of polymorphic definitions can make code maintenance difficult (different objects interpret a method in very different ways)

Examples of Object-Oriented Languages

How many do you know?

Examples of Object-Oriented Languages

How many do you know?

- Java: the best known and most successful
- C++ & STL: the flexibility and efficiency (and some might say obscurity and error-prone features) of C combined with the encapsulation power of objects
- Smalltalk: the first wide-spread object-oriented language
- Eiffel: an object oriented language concerned with verification
- CLOS: common lisp object system (very powerful features including the ability to define your own notions of inheritance, accessors, etc.)
- Many languages support objects: PYTHON, Matlab

Functional Paradigm

- Computation is expressed as **functions** of data

Functional Paradigm

- Computation is expressed as **functions** of data
- In *Pure* Functional Programming there are
  - No explicit assignment or "variables"
  - No explicit control structures such as `IF`, `FOR` or `WHILE`

Functional Paradigm

- Computation is expressed as **functions** of data
- In *Pure* Functional Programming there are
  - No explicit assignment or "variables"
  - No explicit control structures such as `IF`, `FOR` or `WHILE`
- Functional languages are Turing equivalent to procedural languages

Functional Paradigm

- Computation is expressed as **functions** of data
- In *Pure* Functional Programming there are
  - No explicit assignment or "variables"
  - No explicit control structures such as `IF`, `FOR` or `WHILE`
- Functional languages are Turing equivalent to procedural languages
- The key to understanding a functional program is to ask "*What value does it return?*".

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
```

,

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
    ( empty(T) || singleton(T) )  ?
            :
                                                    ,
```

- Find value of condition

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
    ( empty(T) || singleton(T) )   ?
         T    :
                                                ,
```

- Find value of condition
- Empty and single-item lists are already sorted

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
    ( empty(T) || singleton(T) )  ?
         T    :
                                 firsthalf(T) ,
                                 secondhalf(T)
```

- Find value of condition
- Empty and single-item lists are already sorted
- Break up problem and solve pieces

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
    ( empty(T) || singleton(T) )   ?
          T    :
                                        firsthalf(T) ,
                                        secondhalf(T)
```

- Find value of condition
- Empty and single-item lists are already sorted
- Break up problem and solve pieces
  - Partition list 1 into 2 sublists

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
    ( empty(T) || singleton(T) )  ?
          T    :
                          mergeSort(firsthalf(T)),
                          mergeSort(secondhalf(T))
```

- Find value of condition
- Empty and single-item lists are already sorted
- Break up problem and solve pieces
  - Partition list l into 2 sublists
  - Sort each sublist

Functional Sorting

- We could express a sort of set T functionally:

```
S = mergeSort(T) {
    ( empty(T) || singleton(T) )  ?
         T    :  merge(
                     mergeSort(firsthalf(T)),
                     mergeSort(secondhalf(T)))
```

- Find value of condition
- Empty and single-item lists are already sorted
- Break up problem and solve pieces
  - Partition list l into 2 sublists
  - Sort each sublist
  - Merge sorted sublists

Comments on Functional Paradigm I

- New data is computed from old data instead of modifying the old data

Comments on Functional Paradigm I

- New data is computed from old data instead of modifying the old data
- Facilitated by dynamic allocation and garbage collection

Comments on Functional Paradigm I

- New data is computed from old data instead of modifying the old data
- Facilitated by dynamic allocation and garbage collection
- Dominant computational metaphors are
  - composition
  - recursion
    - breaking a problem down into simpler but similar problems
    - solving them and then
    - putting the results back together again

Comments on Functional Paradigm II

- Also known as "Applicative" programming

Comments on Functional Paradigm II

- Also known as "Applicative" programming
- Use recursive structure (e.g. lists and trees)
  - Easy to build from parts created recursively

Comments on Functional Paradigm II

- Also known as "Applicative" programming
- Use recursive structure (e.g. lists and trees)
  - Easy to build from parts created recursively
- Sisal uses compiler tricks and clever datastructures to avoid without copying data repeatedly

Examples of Functional Languages

How many do you know?

Examples of Functional Languages

How many do you know?

- LISP & Scheme (First of its class)
  - was used in AI
  - still used in prototyping and symbolic processing
  - can treat programs as data and data as programs
  - used as a configuration and scripting language
    - CAD/CAM applications and EMACS customizable editor
- ML (non-pure functional language), Haskell (pure)
- Miranda (first functional language intended for commercial applications)

Generic Functions

- Generic functions are to functional languages as class polymorphism is to objected-oriented languages

Generic Functions

- Generic functions are to functional languages as class polymorphism is to objected-oriented languages
- Functions are dispatched based on the types of the arguments supplied to the function

Generic Functions

- Generic functions are to functional languages as class polymorphism is to objected-oriented languages
- Functions are dispatched based on the types of the arguments supplied to the function
- `size-of(list)`, `size-of(vector)` and `size-of(hash-table)` call different underlying implementations

Sort with Generic Functions

- The sort "function" can have different implementations for different types of arguments
  - Integers and reals can be sorted using the ">" partial order relation
  - Vectors could be sorted using their length |V| with a partial order relation
  - Nodes in a graph could be sorted by their degrees
- Again, user doesn't need to understand the details

Languages with Generic Functions

- Common LISP implements generic programming

Languages with Generic Functions

- C++ implement generic programming through the Standard Template Library (STL)
- Common LISP implements generic programming

Declarative Paradigm

- Emphasis is on *what* the computation should achieve - not how

Declarative Paradigm

- Emphasis is on *what* the computation should achieve - not how
  1. Enter *facts* and *rules* (a.k.a. axioms) to describe a situation or domain.

Declarative Paradigm

- Emphasis is on *what* the computation should achieve - not how
  1. Enter *facts* and *rules* (a.k.a. axioms) to describe a situation or domain.
  2. Pose query as a statement to prove

Declarative Paradigm

- Emphasis is on *what* the computation should achieve - not how
  1. Enter *facts* and *rules* (a.k.a. axioms) to describe a situation or domain.
  2. Pose query as a statement to prove
  3. Language searches for a proof of the query

Declarative Paradigm

- Emphasis is on *what* the computation should achieve - not how
  1. Enter *facts* and *rules* (a.k.a. axioms) to describe a situation or domain.
  2. Pose query as a statement to prove
  3. Language searches for a proof of the query
     - The language can return true, false or unproveable

Declarative Paradigm

- Emphasis is on *what* the computation should achieve - not how
  1. Enter *facts* and *rules* (a.k.a. axioms) to describe a situation or domain.
  2. Pose query as a statement to prove
  3. Language searches for a proof of the query
     - The language can return true, false or unproveable
     - The language attempts to find assignments to variables in order to make the statement true

Example Facts, Rules and Queries

- Facts:

```
MATH322 is Boring.
Clyde is an elephant.
```

Example Facts, Rules and Queries

- Facts:

```
MATH322 is Boring.
Clyde is an elephant.
```

- Rules:

```
X is boring ⇒ X makes me sleepy
X is-an elephant ⇒ X is heavy
```

Example Facts, Rules and Queries

- Facts:

  `MATH322 is Boring.`
  `Clyde is an elephant.`

- Rules:

  `X is boring` ⇒ `X makes me sleepy`
  `X is-an elephant` ⇒ `X is heavy`

- Queries:

  `MATH322 is boring` →`true`
  `CMPUT325 is boring` → `unproveable given what you know`
  `There exists an X which is boring`
  →`is true for X = MATH322`

Declarative Sort

- Expressing that S is a sort of set T declaratively:

```
T is-a-sort-of S
  ⇔ T contains each element of S
     and for each element i of T, T(i) > T(i+1)
```

Declarative Sort

- Expressing that S is a sort of set T declaratively:

```
T is-a-sort-of S
  ⇔ T contains each element of S
     and for each element i of T, T(i) > T(i+1)
```

- Given a set of elements T, formulate a statement to prove

$$\exists S. S \texttt{ is} - \texttt{a} - \texttt{sort} - \texttt{of } T$$

Declarative Sort

- Expressing that S is a sort of set T declaratively:

```
T is-a-sort-of S
  ⇔  T contains each element of S
      and for each element i of T, T(i) > T(i+1)
```

- Given a set of elements T, formulate a statement to prove

$$\exists S . S \ \texttt{is} - \texttt{a} - \texttt{sort} - \texttt{of} \ T$$

- Let language search for an S that makes statement true

Declarative Sort

- Expressing that S is a sort of set T declaratively:

```
T is-a-sort-of S
  ⇔ T contains each element of S
     and for each element i of T, T(i) > T(i+1)
```

- Given a set of elements T, formulate a statement to prove

$$\exists S.S \text{ is} - \text{a} - \text{sort} - \text{of } T$$

- Let language search for an S that makes statement true
- The set of possible S's that make the above query true are exactly the legal ways to sort T.

Comments on Declarative Paradigm I

- Dominant computational metaphors are
  - axiomatization (writing down rules and facts)
  - inference

Comments on Declarative Paradigm I

- Dominant computational metaphors are
  - axiomatization (writing down rules and facts)
  - inference
- Sometimes: Easier to say what we want than how to do it
  - But, the computation may be inefficient without constraints on implementation

Comments on Declarative Paradigm I

- Dominant computational metaphors are
  - axiomatization (writing down rules and facts)
  - inference
- Sometimes: Easier to say what we want than how to do it
  - But, the computation may be inefficient without constraints on implementation
- Generic knowledge can sometimes be reused in powerful ways
  - The concept of an ordered set could be used in a sort program, but also reused in reasoning about time intervals or geometric relationships or neighbours

Comments on Declarative Paradigm II

- Correct specification and sound solver implies correct implementation

Comments on Declarative Paradigm II

- Correct specification and sound solver implies correct implementation
- The specification of modules can be composed to create bug free systems at a higher level

Comments on Declarative Paradigm II

- Correct specification and sound solver implies correct implementation
- The specification of modules can be composed to create bug free systems at a higher level
- Declarative knowledge is relational - not functional or causal
  - The statement S $\mathtt{is-a-sort-of}$ T relates S and T
  - We can find a sort S given a set T
  - But, we can also find all sets T that can be sorted to produce S

Comments on Declarative Paradigm II

- Correct specification and sound solver implies correct implementation
- The specification of modules can be composed to create bug free systems at a higher level
- Declarative knowledge is relational - not functional or causal
  - The statement S $\mathtt{is-a-sort-of}$ T relates S and T
  - We can find a sort S given a set T
  - But, we can also find all sets T that can be sorted to produce S
- Unlike functions which always calculate a result from an argument, we say that declarative knowledge can be used in forward or backward directions

Examples of Declarative Languages

- PROLOG (widely used in AI especially in Europe)
  - Did you know that there are object-oriented extensions to Prolog?
  - Implements a limited form of First Order Logic that can be proved efficiently through "resolution"
- SQL (the preeminent language for describing database queries)

Constraint-Based Paradigm

- A restricted form of declarative programming

Constraint-Based Paradigm

- A restricted form of declarative programming
- One defines a set of variables (Item1, Item2)

Constraint-Based Paradigm

- A restricted form of declarative programming
- One defines a set of variables (Item1, Item2)
- One defines domains for variables Item1$\in$\{ a,d,e,f\}

Constraint-Based Paradigm

- A restricted form of declarative programming
- One defines a set of variables (Item1, Item2)
- One defines domains for variables Item1$\in${ a,d,e,f}
- One defines contraints on variables (Item1 $<$ Item 2)

Constraint-Based Paradigm

- A restricted form of declarative programming
- One defines a set of variables (Item1, Item2)
- One defines domains for variables Item1$\in$\{ a,d,e,f\}
- One defines contraints on variables (Item1 $<$ Item 2)
- Language attempts to find a satisfying assignment of variables

Constraint-Based Sorting

- We start with a list $T = (i_1, \ldots, i_n)$ and desire a sorted list $S = (s_1, \ldots, s_n)$

Constraint-Based Sorting

- We start with a list $T = (i_1, \ldots, i_n)$ and desire a sorted list $S = (s_1, \ldots, s_n)$
- Each element of S is a variable which can contain any element of the original list $s_i \in T$.

Constraint-Based Sorting

- We start with a list $T = (i_1, \ldots, i_n)$ and desire a sorted list $S = (s_1, \ldots, s_n)$
- Each element of S is a variable which can contain any element of the original list $s_i \in T$.
- Set up two constraints on each variable s_i
  - No element may contain the same element as another slot $s_i \neq s_j$
  - Each element must have a greater valued entry than its sucessor $\mathrm{val}(s_i) \geq \mathrm{val}(s_{i+1})$

Constraint-Based Sorting

- We start with a list $T = (i_1, \ldots, i_n)$ and desire a sorted list $S = (s_1, \ldots, s_n)$
- Each element of S is a variable which can contain any element of the original list $s_i \in T$.
- Set up two constraints on each variable s_i
  - No element may contain the same element as another slot $s_i \neq s_j$
  - Each element must have a greater valued entry than its sucessor $\mathrm{val}(s_i) \geq \mathrm{val}(s_{i+1})$
- Any satisying assignment of values to variables corresponds to a sort of T

Comments on Constraint Paradigm

- There are often many constraints required to define a problem

Comments on Constraint Paradigm

- There are often many constraints required to define a problem
- Clever techniques can sometimes be used to avoid computing all constraints

Comments on Constraint Paradigm

- There are often many constraints required to define a problem
- Clever techniques can sometimes be used to avoid computing all constraints
- Can do optimization with constraints
  - Common techniques: Linear and Quadratic programs

Probabilistic Inference Paradigm

- An extension of declarative programming

Probabilistic Inference Paradigm

- An extension of declarative programming
- Logics represent uncertainty by disjunction: $a \vee b$, existential quantification: $\exists x.tall(x)$ and negation: $\neg X = fred$

Probabilistic Inference Paradigm

- An extension of declarative programming
- Logics represent uncertainty by disjunction: $a \vee b$, existential quantification: $\exists x.tall(x)$ and negation: $\neg X = fred$
- Probabilistic models represent uncertainty with numbers: $\Pr(a) = \frac{1}{4}$, $\Pr(\neg a) = \frac{3}{4}$

Probabilistic Inference Paradigm

- An extension of declarative programming
- Logics represent uncertainty by disjunction: $a \vee b$, existential quantification: $\exists x.tall(x)$ and negation: $\neg X = fred$
- Probabilistic models represent uncertainty with numbers: $\Pr(a) = \frac{1}{4}$, $\Pr(\neg a) = \frac{3}{4}$
- Can specify conditional probabilities
  - $\Pr(sparrow(\mathrm{aBird})) = 0.80$ - prior probability $\equiv$ fact
  - $\Pr(flies(B)|penguin(B)) = 0$ - conditional probability $\equiv$rule
  - $\Pr(flies(B)|sparrow(B)) = 0.9$

Probabilistic Inference Paradigm

- An extension of declarative programming
- Logics represent uncertainty by disjunction: $a \vee b$, existential quantification: $\exists x.tall(x)$ and negation: $\neg X = fred$
- Probabilistic models represent uncertainty with numbers: $\Pr(a) = \frac{1}{4}$, $\Pr(\neg a) = \frac{3}{4}$
- Can specify conditional probabilities
  - $\Pr(sparrow(\mathrm{aBird})) = 0.80$ - prior probability $\equiv$ fact
  - $\Pr(flies(B)|penguin(B)) = 0$ - conditional probability $\equiv$rule
  - $\Pr(flies(B)|sparrow(B)) = 0.9$
- Language assigns probabilities to statements: $\Pr(flies(\mathrm{aBird})) \rightarrow 0.72$

Comments of Probabilistic Paradigm

- Dominant Constructs
  - Definition of prior and conditional probabilities
  - Probabilistic inference

Comments of Probabilistic Paradigm

- Dominant Constructs
  - Definition of prior and conditional probabilities
  - Probabilistic inference
- Result is a distribution over possible answers
  - $\Pr(\mathit{flies}(\mathrm{aBird})) \rightarrow 0.72$ and $\Pr(\neg \mathit{flies}(\mathrm{aBird})) \rightarrow 0.28$

Comments of Probabilistic Paradigm

- Dominant Constructs
  - Definition of prior and conditional probabilities
  - Probabilistic inference
- Result is a distribution over possible answers
  - $\Pr(\textit{flies}(\mathrm{aBird})) \rightarrow 0.72$ and $\Pr(\neg\textit{flies}(\mathrm{aBird})) \rightarrow 0.28$
- Can be computationally expensive

Comments of Probabilistic Paradigm

- Dominant Constructs
  - Definition of prior and conditional probabilities
  - Probabilistic inference
- Result is a distribution over possible answers
  - $\Pr(\textit{flies}(\text{aBird})) \rightarrow 0.72$ and $\Pr(\neg\textit{flies}(\text{aBird})) \rightarrow 0.28$
- Can be computationally expensive
- Probabilities + utilities →expected values

Comments of Probabilistic Paradigm

- Dominant Constructs
  - Definition of prior and conditional probabilities
  - Probabilistic inference
- Result is a distribution over possible answers
  - $\Pr(\mathit{flies}(\text{aBird})) \rightarrow 0.72$ and $\Pr(\neg \mathit{flies}(\text{aBird})) \rightarrow 0.28$
- Can be computationally expensive
- Probabilities + utilities →expected values
- Choose actions with highest expected values

Concurrent Paradigm

- Many different processes
  All running "at same time"
  Each executing a different instruction

Concurrent Paradigm

- Many different processes
  All running "at same time"
  Each executing a different instruction
- Issues:
  - Allocation of resources
  - Partitioning of computations
  - Communication overhead
  - Synchronization
  - Deadlock, Starvation, . . .

Examples of Concurrency

- Multiplying two $n \times n$ matrices $R = AB$
  - Need to compute n^3 independent values: $R_{ij} = \sum_k A(i,k) \times B(k,j)$
  - Parallelize this to speed up computation

Concurrent Sorting

- The best algorithm for concurrent sorting depends on the architecture of the parallel platform
- For grid processors, we might use a "snake sort"

Paradigm Summary

- Procedural

Paradigm Summary

- Procedural
  - Tell computer to alter data
  - a.k.a. "Imperative"

Paradigm Summary

- Procedural
  - Tell computer to alter data
  - a.k.a. "Imperative"
- Object-oriented

Paradigm Summary

- Procedural
  - Tell computer to alter data
  - a.k.a. "Imperative"
- Object-oriented
  - Extension of procedural
  - Encapsulation provides control over alteration

Paradigm Summary

- Procedural
  - Tell computer to alter data
  - a.k.a. "Imperative"
- Object-oriented
  - Extension of procedural
  - Encapsulation provides control over alteration
- Functional

Paradigm Summary

- Procedural
  - Tell computer to alter data
  - a.k.a. "Imperative"
- Object-oriented
  - Extension of procedural
  - Encapsulation provides control over alteration
- Functional
  - Result is a function of data
  - Data never altered

Paradigm Summary

- Declarative

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers
- Contraints

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers
- Contraints
  - Simplification of logical declarative paradigm

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers
- Contraints
  - Simplification of logical declarative paradigm
- Probabilistic

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers
- Contraints
  - Simplification of logical declarative paradigm
- Probabilistic
  - Declarative paradigm with uncertainty

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers
- Contraints
  - Simplification of logical declarative paradigm
- Probabilistic
  - Declarative paradigm with uncertainty
- Concurrent

Paradigm Summary

- Declarative
  - Define properties of solution
  - Theorem prover finds satisfying answers
- Contraints
  - Simplification of logical declarative paradigm
- Probabilistic
  - Declarative paradigm with uncertainty
- Concurrent
  - Simultaneous execution instructions
  - Requires locking, sychronization, etc.