Programming Languages Prelims Reviewer

Introduction to Programming Languages

Programming Language - communicates instructions to a computer

Reasons to Study Concepts

- increased capacity to express ideas
- improve background for choosing language
- increased ability to learn new languages
- better understanding of the importance of implementation
- better use of already known languages
- overall advancement in computer

Domains

- Scientific applications
- Business applications
- Artificial intelligence
- Systems programming
- Web software

Language Evaluation Criteria

Readability - how easy programs can be understood
Writability - how easy the language can create programs
Reliability - performance under all conditions
Cost

Readability Characteristics

- **Simplicity** - larger number of constructs is harder to learn than less constructs
- **Orthogonality** - ways of combining constructs to build data structures
- **Data Types**
- **Syntax Design** - special words, form and meaning

Writability Characteristics

- **Simplicity and Orthogonality**
- **Abstraction** - ability to define and use structures that allow ignoring of details
- **Expressivity** - convenients ways to specify computation

Reliability Characteristics

- **Type Checking** - testing for type errors
- **Exception Handling** - interception of runtime errors
- **Aliasing** - two distinct names that access the same memory cell
- **Readability and Writability**

Influences on Language Design

Computer Architecture - most languages are designed around Von Neumann Architecture, AKA Imperative Languages

Programming Design Methodologies - evolution of software development methodologies led to new language constructs (Procedural and OOP)

Language Categories

Imperative - updates variables via commands

- Algol, Cobol, PL1, Ada, C, Modula-3

Functional - uses computations as evaluation of mathematical functions, applies functions to values and values are never modified

- Lisp, Haskell, ML, Miranda, APL

Logic - computations as terms of mathematical logic (predicate logic, true or false)

- Prolog

Object Oriented - characterized by **Alan Kay**, everything is modeled as an object, which belong a class and communnicate via message passing

- Smalltalk, Simula, C++, Java

Implementation Methods

Compilation - programs are translated to machine language

Pure Interpretation - uses an interpreter to understand programs

Hybrid - combines compilation and interpretation

Programming Environment

- collection of tools
- UNIX, Jbuilder, Visual Studio, NetBeans

Evolution of Major Programming Languages

Early Languages

Plankakul - program calculus by Konrad Zuse, uses math expressions to show relationships between variables

Pseudocodes - interpretative system for execution

Fortran - *formula translation*, general purpose imperative language for numeirc computation and scientific computing

COBOL - *Common Business Oriented Language* by *Conference on Data System Languages in 1960, used for developing businesses

LISP - List Processor by John McCarthy in the 1950s, first functional language, has Atoms(identifiers) and Lists(lists of atoms), reliant on recursion and has garbage collection

- Scheme - by Guy Lewis Steele Jr. and Gerald Jay Sussman, uses Static / Lexical Scoping (boundaries for variables, like Global Variables vs Local Variables)
- Common LSIP - combines features of several other LISP Dialects, has dynamic scoping

Prolog - Programming Logic by Alain Colmerauer, Phillippe Roussel, Robert Kowalski, consists of collection of statements

ALGOL-based languages

ALGOL - Algorithmic language, for programming scienitfic computations

- Block structure
- Parameter passing
- Control statements
- Recursion
- Dynamic arrays
- Reserved words
- Userdefined data types

BASIC - Beginner's All-purpose Symbolic Instruction Code by John Kemey and Thomas Kurtz, easy to learn and used in terminals connected to a remote computer

PL/I - derived from Programing Language 1, large-scale attempt for a language for various application areas

- concurrent subprogram execution
- handles 23 types of exceptions / errors
- recursive subprograms
- pointers as data type
- referencing of array cross-sections

Pascal - named after Blaise Pascal by Nikalus Wirth, used as teaching tool and beginner's programming language

C - by Dennis Ritchie, for systems programming and implemented in UNIX, contains control statements and rich set of operators

Ada - derived from Augusta Ada Byron, improved code safety and maintainability

- data object encapsulation
- exception handling
- program units can be generic
- concurrent execution of units, tasks

OOP Languages

Smalltalk - by Alan Kay, first to support OOP, uses messages instead of arithmetic / logical expressions

C++ - by Bjarne Stroustrup, can support both OOP and Procedural, overloaded methods, multiple inheritance, and exception handling

- **Objective C** - by Brad Cox (ayo cocks) and Tom Love (ayo Cox-Love?), uses smalltalk syntax to add OOP to imperative language
- **Delphi** - by Anders Hejlsberg, added OOP to Pascal
- **Go** - by Rob Pike, Ken Thompson, Robert GHriesemer, addresses slowness of C++ compilation

Visual Basic - by Bjarne Strostrup, event-driven programming language engineered for type-safe and object-oriented applications

Java - increased level of user-app interaction

- platform-independent
- enchances client interaction
- moves processing to client
- used for scalable internet applications
- publicly available specifications
- enables new forms of software distribution and upgrades

C# - by Microsoft, based on C++, component-based software development

PL Midterms Reviewer

Syntax and Semantics

Basics

Syntax - form of expressions, statements and program units
Semantics - meaning of expressions, statements, and program units
Lexemes - numeric literals, operators, special words
Token - lexeme category

```
index = 2 * count + 17;
```

Lexeme	Token
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
17	int_literal
;	semicolon

Language Recognizer - decides if strings belong to specific language

Language Generator - creates language sentences

Formal Methods of Describing Syntax

Grammar - collection of rules in the language

Noam Chomsky - described 4 classes of generative devices

- **Regular Grammar** - forms and tokens of the language
- **Context-free Grammar** - syntax of the language

Metalanguage - language that defines another language

Backus-Naur Form - natural notation for describing form, by **John Backus** and **Peter Naur** in the 1950s, also uses abstraction for syntactic structures

```
<assign> -> <var> = <expression>
```

Left-hand Side - abstraction being defined

Right-hand Side - the definition of the abstraction defined in the LHS

Rule / Production - mixture of tokens, lexemes, and abstraction references

Nonterminal Symbols - abstractions in a BNF description

Terminal Symbols - lexemes and tokens in a BNF description

Recursive Rule - a LHS that appears in the RHS

Start Symbol - special non-terminal that begins the language sentences

Derivation - sequence of rule applications

- **Leftmost Derivation** - leftmost non-terminal is replaced
- **Rightmost Derivation** - rightmost non-terminal is replaced

Sentential Form - strings in a derivation

Parse Trees - hierarchical representation of the sentences of the language

Ambiguous Grammar - sentential forms with two or more distinct parse trees

Associativity - semantic rule to specify precedence in the event of having two operators with the same precedence

Left Recursive - LHS appearing at the beginning of the RHS

Right Recursive - RHS appearing at the beginning of the LHS

Extended BNF - increases readability and writability of the Backus-Naur Form

- Optional RHS parts are in brackets `[ ]`
  `<if_stmt> -> if(<expression>) <statement> [else<statement>]`
- Braces `{ }` indicate repetition or skipping
  `<ident_list> -> <identifier> {, <identifier>}`
- Element options are placed in parentheses `( )` and separated by an OR operator `|`
  `<term> -> <term> (* | % | /) <factor>`

Attribute Grammar

Static Semantics - rules checked during compilation

Attribute Grammar - describes syntax, static semantics

Attribute Computation Function - specifies computation of attribute values

Predicate Functions - states semantic rules of the language

Classes of Attributes

- **Synthesized Attributes** - passes semantic information up the parse tree
- **Inherited Attributes** - passes semantic information either down or across a parse tree

Dynamic Semantics - rules checked during execution

Semantic Description Methods

- **Operational Semantics** - describes constructs in terms of their effects on an ideal machine
  - **Natural Operational Semantics** - deals with the final result of the execution of a complete program
  - **Structural Operational Semantics** - determines meaning of program through examination of state change sequences
- **Denotational Semantics** - uses mathematical objects to describe meaning of language constructs
- **Axiomatic Semantics** - proves program correctness

Assertions / Predicates - logical expressions in axiomatic semantics

- **Precondition** - assertion before a statement
- **Postcondition** - assertion after a statement

Weakest Precondition - least restrictive precondition and guarantees validity of associated precondition

Inference Rule - inferring of truth of one assertion on the basics of values of other assertions

$$\frac{S1, S2 \ldots, Sn}{S}$$

Antecendent - top part of an inference rule
Consequent - bottom part of an inference rule

Axiom - logical statement assumed to be true

Names, Bindings, and Scopes

Names

Case Sensitivity - design issue for names, along with **relationship of names to special words**, also a problem of readability and writability

Name - string of characters that identify a program entity

Special Words - used to make programs more readable by naming actions to be performed

Keyword - special word that is only special in certain contexts

Reserved Word - special word that cannot be used as a name

Variables

Variable - abstraction of memory cell (or group of cells)

Six Attributes of Variables

- **Name** - identifier, variables that point to the same address are called **aliases**
- **Address** - where the program looks at when it needs to read the data in the variable, also known as *l-value*
- **Value** - data stored in the variable, also known as *r-value*
- **Type** - range of values a variable can store
- **Lifetime** - time when a variable is bound to a memory cell
- **Scope** - range of statements where the variable is visible

Aliasing - allows change of value by assigning it to a different value

Binding

Binding - association between an attribute and an entity, can take place at:

- Language Design Time
- Language Implementation Time
- Compile Time
- Load Time
- Link Time
- Run Time

Static Binding - occrs before runtime and is unchanged throughout execution

Dyanmic Binding - occurs during runtime and can change throughout execution

Explicit Declaration - statement that lists variable names and types

Implicit Declaration - association through default conventions

Dynamic Type Binding - variable is bound to the expression's type

Allocation - taking a memory cell from a pool of available cells

Deallocation - returning an unbound memory cell back to the pool

Variables According to Lifetimes

- **Static Variable** - bound to cells before execution begins, remains bound until execution termination
- **Stack-Dynamic Variable** - storage bindings are created during the **elaboration** of their declaration statements
  - **Elaboration** - allocation and binding of storage indicated by a declaration
- **Explicit Heap-Dynamic Variable** - abstract memory cells allocated and deallocated by explicit run time instructions

- **Implicit Heap-Dynamic Variable** - only bound to heap storage when values are assigned to them

Scope

Scope - refers to where a variable can be referenced in a statement

Scoping Rules

- **Static / Lexical Scoping** - binding names to nonlocal variables
- **Dynamic Scoping** - based on calling sequence and can only be determined at runtime

Other Concepts

Referencing Environment - collection of variables visible in a statement

Named Constant - bound to a value only once, cannot be changed by assignment or input statement

Initialization - binding a value at the time it is bound to a storage

PL Pre-Finals Reviewer

Data Types

Defines a collection of data values and operations

Primitive Data Types - data types not defined using other types

Numeric Types

- **Integer** - most common primitive type
- **Floating-point** - approximations modeled after real numbers
- **Decimal** - fixed number of decimal digits

Boolean - true or false only

Character - stores a single character such as a letter, number, punctuation mark, or whitespace

Character String Types

Consists of sequences of characters

Common operations

- assignment
- concatenation
- substring reference
- comparison
- pattern matching

Length options

- **Static length** - fixed and set when created
- **Limited dynamic length** - can have varying length up to a defined limit
- **Dynamic length** - varying length with no maximum limits

User-defined Ordinal Types

Range of possible values

Types

- **Enumeration** - named constants (possible values) are enumerated in the definition
- **Subrange** - subset of values from another ordinal type

Structured Data Types

Array - container object that stores a fixed number of values of a single type

Subscript / Index - used to refer to values in an array

Array categories

- **Static** - index and storage allocation is static
- **Fixed stack-dynamic** - index is static, but allocation is done at declaration elaboration
- **Stack-dynamic** - index and allocation are dynamic at elaboration
- **Fixed heap-dynamic** - index and storage binding are fixed after storage allocation
- **Heap-dynamic** - index and storage allocation are dynamic and can change multiple times

Array operations

- assignment
- concatenation
- comparison
- slices

Array types

- **Rectangular array** - multidimensional array with equal rows and columns
- **Jagged array** - has rows with different number of elements
- **Slice** - extracted array from another array
- **Associative array** - unordered collection with keys and values

Record - collection where elements are identified by names and access through offsets

Tuple - Similar to records, but does not have named elements

Lists - ordered sequence of values usually not separated by punctuation

Union - variables that can store different types during program execution

- **Free Unions** - unions that are free from type checking
- **Discriminated Union** - ~~black lives matter~~ unions that have a discriminant, a tag that tells the union's type indicator

Pointer and Reference Types

Pointer Type - range of values that are memory addresses and a value called *nil*

Nil - indicates that a pointer cannot be used to reference a memory cell

Heap - area where storage is dynamically allocated

Heap-Dynamic Variables - dynamically allocated variables

Anonymous Variables - variables without names

Pointer operations

- Assignment - set value to an address
- Dereferencing - takes reference through one level of indirection

Reference type - object or value in memory

Other Concepts

Type Checking - ensures types of operands and operators are the same

Coercion - conversion of an operator

Type Error - application of error to an operand of another type

Dynamic Type Checking - type checking at run time

Strongly-typed Programming Language - a language where type errors are always deteced

Type Equivalence - two types that don't produce an error and can proceed with the expression without coercion if swapped with the other type

- Approches to type equivalence
  - name type equivalence
  - structure type equivalence

Expressions and Assignment Statements

Arithmetic Expressions

- specifies computations; consists of operators, operands, parentheses, and operation calls
- Types of operators:
  - Unary - single operand
  - Binary - 2 operands
  - Ternary - 3 operands
- Binary operators are **infix** in most languages

Operator Precedence - specifies order of operations

Precedence	Ruby	C-Based Languages
Highest	**	postfix ++, --
	unary +, -	prefix ++, --, unary +, -
	*, /, %	*, /, %
Lowest	binary +, -	binary +, -

Adjacency - operators separated by a single operand

Associativity - evaluates operators with the same precedence

- Left Associativity - left operator evaluated first
- Right Associativity - right operator evaluated first

Language	Associativity Rule
Ruby	Left: *, /, +, - Right: **
C-Based Languages	Left: *, /, %, binary +, binary - Right: ++, --, unary -, unary +
Ada	Left: all except ** Nonassociative: **

Parentheses - can alter precedence and associativity

Side Effect - when a function changes one of its parameters or a global variable

Global Variable - variables declared outside the function

Referential Transparency - two expressions that can be substituted for one another without affecting the program

Operator Overloading

- operators can be used for more than one purpose

Type Conversions

Narrowing Conversions - conversion to a type that cannot store even approximations of the value

Widening Conversions - conversion to a type that can include at least approximations of the value

Mixed-Mode Expression - expressions with differently typed operands

Cast - explicit type conversion `string phone = (string)number`

Overflow/Underflow - result of an operation is either too large or too small

Relational and Boolean Expressions

Relational Operator - compares values of 2 operands

Relational Expression - 2 operands and 1 relational operator

Boolean expressions - consists of boolean variables, boolean constants, boolean operators, and relational expressions

Short Circuit Evaluation

- expression whose result is determined without evaluating all operands

Assignment Statements

Assignment Statement - stores a value in a variable

Compound Assignment Operator - combination of an assignment operator and an arithmetic operator ex. `sum += value`

Prefix/Postfix Operators - found at the beginning of end of the operand

Mixed Mode Assignment - assignment with different data types

Statement-Level Control Structures

Control Statements - alters execution path of the program based on conditions

Control Structure - control statement and the other statements that it controls

Categories of Control Statements

- selection
- multiple selection
- iterative
- unconditional branching

Selection Statements

provides means of choosing between 2 or more execution paths

General two-way form (pseudocode):

```
if control_expression
  then statement
  else statement
```

General n-way form (pseudocode):

```
switch(control)
  case 1: statement
  case 2: statement
  case n: statement
  default: statement
```

`default` **clause** - for unrepresented values

`break` **clause** - transfers control out of the switch

`goto` **clause** - transfers control to other selectable statement

Iterative Statements

Loop, causes statements to execute over and over

Pretest/Posttest - the condition testing of a loop that appears either before (pre) or after (post) the loop body

Loop variable - counting variable of loops

Stepsize - difference of sequential loop variables

Loop parameters - initial, terminal, stepsize

Logically controlled loop - loops based on Boolean expressions

User-located control loops - loop controls that the programmer decides besides on the top or bottom of the loop body

`continue` **clause** - skips rest of loop on current iteration

Data-based Iterators - iteration depends on the number of elements ex. foreach

Unconditional Branching

Transfers execution control to a specified location in the program (goto)

PL Finals Reviewer

Subprograms

Fundamentals

Subprogram - program called by another program to perform a specific task, building blocks of programs

Characteristics of Subprograms

- single entry point
- only one subprogram in execution at a time
- control returns to caller at the end

Subprogram Definition - describes interface and actions of subprogram

Subprogram Call - explicit request to execute

Subprogram Header - includes name, type, and parameters

Actual Parameters - parameters used in the call (arguments)

- **Positional Parameter** - relies on the sequencing of arguments

```
showOutput("Hello World!", 5)
```

- **Keyword Parameter** - uses keywords to define arguments

```
showOutput(msg: "Hello World!", time: 5)
```

Default Value - value of a parameter when no argument is passed

Kinds of Subprograms

- **Procedures** - user-defined computation statements
- **Functions** - mathematical functions

Local Referencing Environment

Association - binding of identifiers to objects and subprograms

Referencing Environment - set of identifier associations

Local Referencing Environment - set of associations that are defined within the subprogram

Local Variables - variables defined inside a subprogram

- **Static** - bound to memory cell before execution
- **Stack-dynamic** - bound to storage at execution and unbound at termination

Parameter Passing

Parameter Passing Method - mechanism of passing arguments to a subprogram

Semantic Models of Formal Parameters

- **In mode** - receives data from corresponding actual parameter
- **Out mode** - transmits data to acutal parameter
- **Inout mode** - can do both

Parameter-passing Methods

- **Pass-by-Value** - arguments are used to initialize corresponding parameter
- **Pass-by-Result** - no value transmitted
- **Pass-by-Value-Result** - combination of above methods
- **Pass-by-Reference** - transmits access path or address to subprogram
- **Pass-by-Name** - argument is textually submitted

Parameters as Subprograms

- **Shallow Binding** - environment of call statement that executes it
- **Deep Binding** - environment of the passed subprogram definition
- **Ad Hoc Binding** - environment of call statement that passed the subprogram as argument

Other Concepts

Overloaded Subprogram - same name as another subprogram but different signature

Polymorphic Subprogram - takes different type parameters on different activations

Ad Hoc Polymorphism - polymorphism provided by overloaded subprograms

Subtype Polymorphism - type variable that can access any object with the same type or any type derived from it

Parametric Polymorphism - subprogram that takes generic parameters, AKA **generic subprograms**

Closure - subprogram and its referencing environment

Coroutine - special program with multiple entries

Abstraction

Data Abstraction

Abstraction - view of an entity that only includes the most significant parts

- **Process Abstraction** - specifies processes without the details of how it is performed
- **Data Abstraction** - represents the needed information in a system

Abstract data type - only includes representation of data type and subprograms related to it

Object - instance of abstract data type

User Defined Abstract Data Types

- object representations are hidden and only allows direct operations provided in the definition
- declarations and protocols are contained in 1 syntactic unit

Benefits of Information Hiding

- clients cannot manipulate representations of objects
- reduces range of code and number of variables
- reduces name conflicts
- simplifies modification and repair

Accessor Methods - getters and setters, methods that return and set private variables

Advantages of Accessor Methods

- read-only access with getters and no setters
- support for constraints
- implementation of data member can be changed without affecting clients

Support of Abstract Data Types

C++

- **Structs for Abstract Data Types**
    - **Class** - reference type
    - **Struct** - value type
- **Data Members** - data in a class
- **Member Functions** - functions in a class
- C++ can have constructors and destructors
- C++ hides entities through the `private` clause

Java

- objects in java are allocated from the heap and accessed through reference variables
- **Stack** - last in, first out with the operations below:
  - **push** - places item to top of stack
  - **pop** - removes item and returns it

C#

- provides properties to implement getters and setters without explicit method calls

Ruby

- classes are declared with `class` and closed with `end`
- instance variables begin with `@`
- instance methods begin with `def` and closed with `end`
- class methods are distinguished from instance methods by having class names appended at the beginning `className.method`
- constructors are named `initialize`

Concurrency

Fundamentals of Concurrency

Levels of Concurrency

- **Instruction Level** - two machine instructions executed at the same time
- **Statement Level** - two high-level langauge statements executed at the same time
- **Unit Level** - two subprogram units executed at the same time
- **Program Level** - two programs executed at the same time

Categories of Concurrency

- **Physical** - multiple processors used
- **Logical** - only one processor used

Thread of Control - sequence of program points

Multithreading - programs with more than one thread of control

Benefits of Concurrency

- machines with multiple processors can more effectively execute programs
- programs written for concurrent execution is faster than those with sequential execution
- provides method of conceptualizing program solutions to problems
- synchronization of programs in different machines

Subprogram-Level Concurrency

Task - AKA Process, unit of program that can be concurrent with other units of the same program

Threads - methods that serve as tasks and executed in objects

Characteristics of Tasks

- can be implicitly started
- programs do not need to wait for a task to complete to continue
- control may or may not return to the caller when a task execution is done

Categories of Tasks

- **Heavyweight** - execute in own address space
- **Lightweight** - executes in the same address space as other tasks

Disjoint - task that does not communicate with other tasks

Synchronization - mechanisms that controls order of task execution

- **Cooperation Synchronization** - 1 task must wait for another before continuing
- **Competition Synchronization** - both tasks require the same resource that cannot be used simultaneously

Scheduler - manages sharing of processors among tasks

States of Tasks

- **New** - created but not yet executed
- **Ready** - ready to run but not running yet
- **Running** - currently executing
- **Blocked** - running task that has been interrupted
- **Dead** - no longer active

Semaphores

Approaches to Language Support for Concurrency

- Semaphore
- Monitor
- Message Passing

Semaphore - consists of an integer and a queue

Task Descriptor - sotres all relevant information about the task execution state

Operations

- **Wait** - decrement semaphore value, negative values block the executing process

- **Release** - increment semaphore value, negative values unblock a blocked process

Monitors

Monitors - provides natural way of providing mutually exclusive access to data among tasks

Components

- local data
- condition variables
- procedures to access data and condition
- intialization code
- implicit queues

Characteristics

- local data variables are only accessible by the monitor
- processes may enter the monitor by invoking its procedures
- only 1 process can execute a monitor procedure at a time

Message Passing

Message Passing - tasks communicate with each other to synchronize

Operations

- Send
- Receive

Types

- **Synchronous** - processes that run at the same time
- **Asynchronous** - processes that are not running at the same time

Java Threads

Thread - program execution path, AKA Lightweight Processes, each Java program starts with a main thread

Ways of Creating a Thread

- Provide `Runnable` object
- Extend `Thread` class

Methods

- `run()` - performs action for thread
- `start()` - causes thread to begin execution

- `sleep()` - suspends thread for a certain amount of time
- `join()` - makes thread wait for the completion of another thread
- `interrupt()` - stop thread action
- `setPriority()` - change thread priority
- `setName()` - change thread name
- `isAlive()` - checks if thread is still alive

`java.util.concurrent` - built-in library of classes for concurrent programming

C# Threads

Uses `System.Threading` namespace

Creating a thread

```
System.Threading.Thread newThread = new System.Threading.Thread(AMethod);

//start
newThread.Start();

//stop
newThread.Stop()
```

Methods

- **Start** - begin execution
- **Sleep** - pause thread
- **Suspend** - pause thread when it reaches safe point
- **Abort** - stop thread
- **Resume** - restart suspended thread
- **Join** - waits for another thread to finish