CS3723 Programming Languages Overview

|  |  |
| --- | --- |
| **What is the best programming language?**  **Is there a programming language that is suitable to all problems being solved?** |  |
| **Why Study Programming Languages**   1. You will encounter many languages. This course will help you more quickly learn new languages. | Most of you will be exposed to over 30 programming languages in your lifetime. In your professional career, you will encounter a new language every 1-3 years. My examples   * Primary languages: FORTRAN, PL/I, COBOL, C, C++, Java, Visual Basic, Python * Languages in academia: ALGOL, Pascal, SNOBOL * Artificial intelligence: LISP * Database: DL/I, DBTG, SQL, EZTrieve, ACE * Text formatting: Scribe, Script, troff * Shell & scripting: bash, tcsh, Perl * proprietary languages: Q++, CCS Text formatting, EEL, uniform rules * assembler: IBM s/370, PDP 11, IA32 * others: HTML, GPSS, XML |
| 2. You may need to define a language. With this course and CS 4713 Compiler Construction, you can more rapidly design special purpose languages. | Suppose you need to simplify how business administrators define and implement business rules (or something else).   * What language syntax would the business administrators use? * What language constructs (if-then-else, while, for, case) would help? * What would be the runtime form for executing the language? |
| 3. Better way to express ideas. | When you took data structures, you learned that certain algorithms are better expressed using stacks. Your use of push, pop, and isEmpty improved how you could conceptualize an algorithm (e.g., converting infix expressions to postfix). Certain language features make it easier to express the ideas necessary to solve particular type of problems.  Uniform Rules allowed a large insurance company to externalize most of their auto insurance business rules, saving $300m / year. |
| 4. Better understanding of how to pick a language for a problem. | Suppose you need to create a GUI. Of C, C++, and Java, what would you (not) pick?  Suppose you needed to do a scientific application requiring many calculations, what language would you pick?  Suppose you needed to write some simulations before a business implements their ideas, what would you pick?  Suppose you needed a program that understands natural language, what language would you pick? |
| 5. Better understanding of programming languages will improve your code and reduce bugs | What is the appropriate implementation of error handling in your program?  What can you do in your code to improve maintainability?  Suppose we want to produce the reverse of a string in Java. What is good/bad about this approach?  static String reverse(String str)  {  String rev = "";  for (int i = 0; i < str.length(); i++)  rev = str.charAt(i) + rev;  return rev;  }  String origStr = "abcdef";  String revStr = reverse(origStr);  ?? |
| **Terminology for Programming Language Categorization**  **Imperative** command-oriented statements that change a program's variables  **Structured** imperative without GOTO statements  **Procedural** a step-wise structured process where the program specifies **how** to do something  **Functional** uses functions which given a particular value of X always returns the same result.  This means that there are no side effects.  **Object-oriented** software is divided into classes which contain data and are only manipulated by a set of methods.  **Logical** Also known as nonprocedural. The program specifies **what** is needed instead of how to do it.  What is the advantage of a program written using a functional language? ??  What is the advantage of a nonprocedural language?  ?? | Imperative FORTRAN (uses GOTO statement)  Procedural COBOL, PL/I, C, Python  Functional LISP (although most implementations provide nonlocals)  LISP examples:  (COND ( (> X Y) X )  ( T Y ) )  (+ (\* L 2) (\* W 2))  Object-oriented C++, Java, Python  Logical SQL, Prolog  SQL example:  select Student.Id, Student.Name  from Student, Enrollment  where Student.Id = Enrollment.Id AND  Enrollment.CourseNr = "CS3723"; |
| **Programming Language Considerations**  These considerations greatly impact a programming language and your selection of a language  **Programming Domain**  **Ease of Use and Maintainability**  **Data**  **Data Control**  **Parameter Passing**  **Translation and Execution**  **Operations and Sequence Control**  **Binding**  **Abstractions and Object Orientation**  **Error Handling** |  |
| **Programming Domain**  Particular programming domains influence programming languages.  **Scientific**  **Business**  **Natural Language**  **Data Science**  **Web UI** | Scientific   * FORTRAN was one of the first programming languages that is still around today * C is available on most platforms and is replacing FORTRAN   Business   * COBOL was one of the first programming languages and is still used today. Good decimal integration. * PL/I was primarily available on IBM platforms. Good decimal integration * Java is being used for many business applications today although it doesn't have good decimal integration, structures nor record I/O   Natural Language (and Artificial Intelligence)   * LISP mostly uses linked lists and recursion. Data and code are represented similarly. * Python has some capabilities for handling language patterns.   Data Science   * Structured Query Language (SQL) is the most popular query language * SAS tools are widely used by data scientists * Matlab has popularity for doing scientific and engineering reporting applications.   Web UI   * HTML * JavaScript * PHP |
| **Ease of Use and Maintainability**  A programming language is a tool to help you specify the solution to problems. As you have learned, different tools make it easier to solve certain types of problems. For example, Microsoft Excel is very good at helping with problems involving money. With its query capabilities, it has a wider range of possibilities.  It is important that a language makes it easier to express a solution to a problem. This improves ease of use and maintainability. Corporations prefer that your code be easy to understand and for you to avoid *tricks.* | The "=" operator is used for equivalence testing in some languages. Some languages allow "=" to be an intermediate assignment operator within expressions. This can cause confusion.  Examples: Assume a = 3 and b = 5. In C, this is true:  if (a = b)  blah;  Sometimes, language designers stimulate poor ease of use and maintainability:  i<lim-1 && (c = getchar()) != '\n' && c != EOF // Chapter 2 KR  while (\*s++ = \*t++) // Chapter 5, KR  ;  Examples taken from *The C Programming Language* by Kernighan and Ritchie. |
| **Data**  In programs, the values (i.e., data) of variables change allowing programs to be able to handle multiple sets of data.    Characteristics of data:  **Location** where it is located  **Data type** describes acceptable values (char, integer, float, boolean)  **Structure** primitive, homogeneous array, record structure; self-referencing; attributes for size and value  **Size** size can be in bits or bytes; fixed or variable length; lower bounds and upper bounds  **Value** can vary or it can be immutable; the actual value  Additionally, we have the concept of a **descriptor** which describes the data, and often includes data type, structure, and size. Almost all languages have descriptors during translation. A language can be less efficient (time) if descriptors are necessary during execution.  Strings are represented differently in various programming languages. (Please see the examples to the right.)  What are the advantages/disadvantages of how C represents strings?  Disadvantages:   * ??   Advantages:   * ??   In Java, a String variable's value can be changed; however, an actual string is immutable (i.e., cannot be changed). Why?  ?? | Representing strings varies in languages  COBOL - fixed-length; parameters must be declared with same size  01 STUDENT.  02 ABC123 PIC X(6).  02 NAME PIC X(30). \*> padded on right with spaces.  PL/I - fixed-length or variable-length (size and value within a declared max size); parameters can receive descriptors (specifying max size and location)  DCL ABC123 CHAR(6),  NAME CHAR(30) VARYING;  DCL NAME CHAR(\*) VARYING; /\* receives a descriptor \*/  C - variable-length (marker); parameters don't know maximum size.  char szName[31] = "Joe King"; // C uses a zero byte for markers  C++ - variable-length (current length, allocated size (most implementations), location)  std::string s1 = "Lee King";  std::string s2("Rea King");  Java - immutable (String class) has size, offset, and location; char arrays allow changes  String name = "Ray";  name = name + " King";  char nameChArray[] = "Telly Phone".toCharArray();  nameChArray[6] = 'G';  nameChArray[7] = 'r';  nameChArray[8] = 'a';  nameChArray[9] = 'p';  nameChArray[10] = 'h';  Python - immutable has size and location;  name = "Faye";  name = name + " King"; |
| **Data Control**  These are features of a programming language managing the accessibility of data during different points in program execution.  **Referencing Environment -** the set of currently active associations (includes global, current locals, current parameters, and current non-locals)  **Static vs Dynamic Scope** - based on language definition, non-local variables can be associated based on dynamic call sequence or on static physical structure of the code | // the result of this code will be different for dynamic vs static scope  // using C-like syntax  int x = 10;  int y = 20;  funcA();  void funcA ()  {  int y = 50;  funcB();  printf("A: %d %d\n", x, y);  }  void funcB()  {  x += 5;  y += 5;  printf("B: %d %d\n", x, y);  }  Output with static scope (non-locals are in surrounding static code structure):  B 15 25  A 15 50  Output with dynamic scope (non-locals are impacted by calling sequence):  Recent call is called!>  B 15 55  A 15 55 |
| **Parameter Passing**  **Arguments (i.e., actual parameters) -** passed to a function. Arguments might be identifiers, constants or expressions (which might involve function calls)  **Parameters (i.e., formal parameters) -** represent the arguments in the called function  Parameter transmission techniques:  **By Value -** the value of the argument is passed and becomes the value of the formal parameter  **By Reference -** *conceptually* a pointer (usually the location of the argument) is transmitted; function can modify the argument; often called **by address** parameter passing  **By Name -** transmit an unevaluated argument, allowing the called function to evaluate it  Note: many languages use **By Value**, but pass references to objects (i.e., the pointer's value is copied not the address of the variable). You will find many incorrect sites which incorrectly state a language has by reference parameter passing since it passes a reference. It is better to call this **by value object reference**. | C   * arrays are passed by reference * by default, everything else is passed by value; however, an address of the argument can be passed by using the & operator   // call  determineMinMax(gradeM, iNumEntries, &dMin, &dMax);  // function declaration  void determineMinMax(double gradeM[], int iNumEntries  , double \*pdMin, double \*pdMax)  {  …  if (gradeM[i] > \*pdMax)  \*pdMax = gradeM[i];  …  }  PL/I   * uses **by reference** * instead of the programmer dereferencing a parameter in the called function, the compiler did that under the covers.   /\* call \*/  determineMinMax(gradeM, iNumEntries, min, max);  /\* function declaration \*/  determineMinMax: PROC (gradeM, iNumEntries, min, max)  DCL gradeM(\*) FLOAT,  iNumEntries FIXED BIN,  min FLOAT,  max FLOAT;  …  IF gradeM(i) > max  max = gradeM(i);  …  END determineMinMax;  Java   * uses by value and by value object reference * when an argument is a reference to an object, a copy of that reference is passed. (This is not by reference parameter passing.) |
| **Translation and Execution**  This topic is frequently known as **compilation** vs **interpretation**.  Many traditional languages (COBOL, FORTRAN, PL/I, C) have most of these steps in translation:  **Preprocess -** converts preprocessor directives to source language statements  **Compile** - generates assembly language from the source language code  **Assemble** - generates machine instructions for the assembly language statements  **Link** - resolves external references to create an executable  Another approach is to do all translation/execution at runtime. With **interpretation**, the language is quickly parsed and executed instead of having formal compilation steps prior to execution.  Java uses a hybrid of compilation and interpretation. | C (PL/I has the same steps)   * Preprocessor translates #include, #define, #ifdef, #ifndef directives into their corresponding C code * Compiler generates appropriate assembly language (depending on hardware and operating system) * Assembler generates machine code * Linker resolves external references (e.g., global extern references to global extern basis, function calls to the address of those functions)   LISP   * No preprocessor, compiler, assembler, linker * Interpreted at runtime from a prefix syntax using a LISP Virtual Machine   Java   * No preprocessor * Compiler generates appropriate bytecodes (non-architecture specific assembly-like language) * Bytecodes are interpreted at runtime using a Java Virtual Machine * When necessary for performance, some implementations will translate to machine code prior to execution or dynamically during execution |
| **Operations and Sequence Control**  These provide the essential framework within which operations and data are combined.  Some characteristics of operations:  syntax notation (prefix, infix)  precedence  extensibility  number of operands  explicit vs implicit operations  explicit vs implicit operands  explicit vs implicit results  invariant vs generic | Infix notation:  perimeter = 2 \* width + 2 \* length; // rectangle  perimeter = l1 + l2 + l3 + l4; // four-sided polygon  What are some shortcomings of infix notation?   * ??   How would we represent a substring operation which requires 3 operands (string, start, length)?  ??  Prefix notation in LISP:  (+ (\* 2 width) (\* 2 length))  (+ l1 l2 l3 l4) |
| **Sequence Control**  In most languages, flow is linearly downward statement by statement. We have statements which can alter that flow:  Conditional - if, if-then-else, case constructs  Iteration - loop, repetition (e.g., for)  Subprogram - macros, functions, recursion, parallel, event-based  Interrupts - an unusual event altering flow | PL/I case statement examples:  /\* Example 1 \*/  SELECT (TRANSACTION.COMMAND);  WHEN ('DEPOSIT') CALL DEPOSIT(TRANSACTION, ACCOUNT);  WHEN ('WITHDRAWAL') CALL WITHDRAWAL(TRANSACTION, ACCOUNT);  WHEN ('INTEREST') CALL ADD\_INTEREST(TRANSACTION, ACCOUNT);  OTHERWISE  DO;  PUT SKIP EDIT('INVALID COMMAND: ', TRANSACTION.COMMAND)  (A, A);  END;  END;  /\* Example 2 \*/  SELECT;  WHEN (TRANSACTION.COMMAND = 'WITHDRAWAL'  & ACCOUNT.BALANCE < 0) CALL OVERDRAWN(TRANSACTION, ACCOUNT);  WHEN (TRANSACTION.COMMAND = 'DEPOSIT') CALL DEPOSIT(TRANSACTION, ACCOUNT);  OTHERWISE CALL WITHDRAWAL(TRANSACTION, ACCOUNT);  END; |
| **Binding**  The point when a program element is bound to a characteristic or property strongly influences a programming language.  **Categories of Binding Times**  **Execution Time** - during program execution  **Translation Time** (compile time) - bindings performed by the compiler  **Language Definition Time -** when the language was described  **Language Implementation Time -** variations for a particular implementation of the language | Execution Time   * binding of variables to values * binding of variables to their locations (e.g., automatics in C) * binding of parameters to arguments   + on function entry for **by value** and **by address** parameters   + on reference for **by name** parameters   Translation Time   * binding of variables to their data types (C) * binding of variables to their structure (C) * binding of variables to their locations (static in C, but the actual address is bound by the loader)   Language Definition Time   * meaning of +, -, \*, / * meaning of if, printf   Language Implementation Time   * ordering of bits (little endian, big endian) * size of int, size of long * meaning of isalpha in C |
| **Object Orientation**  The abstraction and object orientation capabilities of a language can impact the extensibility of the language.  Many languages (COBOL, PL/I, C) allow the definition of abstractions in the form of record structures. A student record structure can be defined to have many attributes. Since a programmer can reference the higher concept of a student, this helped simplify understanding (and ease of use).  With object orientation, we go beyond simply data. We can also define the operations that are allowed on it. For example, we can define the operations admitUnderGraduate, admitMasters, enroll, and withdraw as operations specifically for a student.  With modularity (instead of using OO methods), we can also create similar functions, but we have to pass in the structure. This exposes the structure to the outside. | **// C typedef for Student**  typedef struct  {  char szABC123[7];  char szFirstNm[30];  char szLastName[30];  char szMajorCd[4];  int iGradePoints;  int iGradeHours;  char cClassification;  char cStatus;  } Student;  **// Java Student Class**  public class Student  {  String abc123Id;  String firstNm;  String lastNm;  String majorCd;  int gradePoints;  int gradeHours;  char classification;  char status;  public Student()  {  classification = 'U'; // unknown  status = 'U'; // unknown  }  public void admitUnderGraduate()  {  classification = 'F'; // freshman  status = 'A'; // active  } |
| **PL/I - everything including the kitchen sink in the language definition** | PL/I provides all these capabilities as part of the language definition:  record structures  arrays, multi-dimensional arrays, LBOUND, HBOUND, array slices  Strings, substring, concatenation (via || operator), string comparison (via = operator)  Bitstring, substring, concatenation (via || operator)  stacks (via CONTROLLED storage)  integer, floating point, packed numeric data  parallel tasks  PL/I had proprietary compilers (not open source).  PL/I did not have OO.  Although very popular in organizations using IBM mainframes, it didn't have wider appeal. Why?  ?? |
| **Error Handling**  The ease of detecting and handling errors influences programming languages.  **Interrupts -** an unusual event, detectable by software or hardware, that may require special processing.  **Exceptions** - events generated by interrupts or other software  Some languages (e.g., C++, Java) allow programmer-defined exceptions.  Some languages provide extra features to aid in fixing error situations. Visual Basic and PL/I allow control to be returned back to the point of where an error occurred. | Interrupts   * divide by zero * invalid address reference (segmentation faults) * page fault * break points * arithmetic overflows/underflows   Software-generated Exceptions   * stack underflow (popping an empty stack) * subscript range |