CS345 - Programming Languages

Lecture 02

Dr. Greg Lavender
Fall Semester 2007
Department of Computer Sciences
University of Texas at Austin

"Syntactic sugar causes cancer of the semi-colons"

-- Alan Perlis, Turing Award Winner

What Is a Programming Language?

- Common Answers
  - A set of imperative statements/commands used to direct a computer to do something useful.
    - for example, in C, we print to an output device:
      - `printf("hello world\n");`
      - what function (in the mathematical sense) is "computed" by the procedure printf?
  - A formal "notation" for specifying computations, independent of a specific machine.
    - the purpose of the machine is to execute the computations specified in our language notation.
      - for example, a factorial function takes a single non-negative integer argument and computes a positive integer result.
      - mathematically, the type of this function is written as:
        - *fact*: *nat -> nat*

What Is a Programming Language?

- The factorial function type declaration does not convey how the computation is to proceed.
  - we also need a computation rule:
    - *fact (0) = 1*
      *fact (n) = n * fact (n-1)*
- This notation is more computationally oriented and can almost be executed by a machine.
  - compare this notation to the following programming language definitions for a factorial function

Factorial Functions

- C, C++, Java:
  - ```
    int fact (int n)
     { return (n == 0) ? 1 : n * fact (n-1); }
    ```
- Scheme:
  - ```
    (define fact
        (lambda (n)
            (if (= n 0) 1 (* n (fact (- n 1))))))
    ```
- ML:
  - ```
    fun fact n = if n=0 then 1 else n*fact(n-1);
    ```
- Haskell:
  - ```
    fact :: Integer->Integer
    ```
  - ```
    fact 0 = 1
    fact n = n*fact(n-1)
    ```
    ```

Why So Many Languages?

- Computer Scientists are mainly interested in building automated software tools that help people cope effectively with *complexity*.
  - failure to manage complexity can result in catastrophic errors. A small coding mistake can kill somebody!
  - in programming, every line of code in a program is a potential single point of failure for the system.
- Complexity usually arises because of large scale or “hard” problems.
  - The sheer number of logic components in a modern CPU requires thousands of simulations to check for problems, but that doesn’t mean they are all found.
  - Differential cryptanalysis is concerned with factoring large integers, requiring hundreds or thousands of machines cooperating on the solution
    - E.g., the breaking of the Data Encryption Standard (DES) using hundreds of machines on the Internet.

Why So Many Languages?

- The one thing you can count on is that in your lifetime there will be new programming languages invented
  - you will likely have to spend time learning them on your own!
  - But for now, you have the luxury of being able to take a class.
- There are often conflicting goals for a language, which can lead to “feature creep”.
  - the result is often that a simple, and perhaps elegant language ends up hideously more complex than was intended (e.g., PL/I and C++).
  - this sometimes causes someone to get fed up, and invent a language that adopts the original simple and elegant ideas, while eliminating the complexity (e.g., Java).

Why So Many Languages?

- Computer Scientists are always looking for better ways to build software tools to deal with various computationally solvable problems.
  - many programming languages are general purpose tools
  - others are targeted at solving specific kinds of problems (e.g., massively parallel computations).
  - there have been literally hundreds of different programming languages designed, some general purpose, others special purpose. Many are now obsolete.
- The trend that keeps repeating itself is that useful ideas evolve into new language designs
  - Algol -> Simula -> SmallTalk -> C with Classes -> C++
  - languages are also invented for special purpose reasons
    - E.g., scripting languages: Perl, Tcl, Python, Ruby, etc.

CS345: Lecture 02 - Programming Paradigms

Algol-like Languages

- Many modern programming languages are descended from Algol-60
  - Algol was the first language to introduce the idea of **lexical scoping**
    - begin…end or { … }
    - enabled the definition of modular procedures and the expression of recursive functions
- Algol influenced many of today's languages
  - Scheme is a dialect of Lisp, but claims Algol as its conceptual ancestor because of the use of lexical scoping. But the syntax is like LISP.

Programming Paradigms

- The principal paradigms are:
  - Imperative/Procedural Programming
  - Functional/Applicative Programming
  - Object-Oriented Programming
  - Concurrent Programming
  - Logic Programming
  - Scripting Language Programming

Programming Paradigms

- “Paradigms” emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas.
  - the term paradigm comes from the book, “The Structure of Scientific Revolutions” by Thomas Kuhn.
- Programming paradigms are the result of people’s ideas about how programs should be constructed
  - and the development of *formal linguistic mechanisms* for expressing those ideas
  - and software engineering principles and practices for using the resulting programming language to solve problems.

Programming Paradigms

- Imperative (procedural) programs consists of actions to effect state change, principally through assignment operations or *side effects* (e.g., pass-by-reference)
  - Imperative languages: Fortran, Algol, Cobol, PL/I, Pascal, Modula-2, Ada, C
  - why does imperative programming dominate in practice?
- *OO* programming is not always imperative programming
  - but most *OO* languages have been imperative: Simula, Smalltalk, C++, Modula-3, Java.
    - however, CLOS (Common Lisp Object System) is an object-oriented version of Lisp which is not imperative

Programming Paradigms

- Not all functional languages are "pure"
  - Pure means no "side effects"
    - Side-effects are a result of the assignment statement.
  - practical functional programming languages rely on non-pure functions for input/output and some permit assignment like operators
    - E.g, (set! x 1) in Scheme
- Concurrent programming cuts across imperative, object-oriented, and functional programming.
- Logic programming is based on predicate logic
  - targeted at theorem proving languages and database applications
- Scripting is a very high-level of programming
  - often used for rapid development
  - used to "glue" together different programs
  - used to script active web pages
  - often *dynamically typed* having only int, float, string, and array as the data types. No user defined types
  - variables are *weakly typed*, so a variable 'x' could be assigned any type at any time during execution

Unifying Concepts

- The unifying concepts in languages are:
  - Types (both builtin and user-defined)
  - Expressions (e.g., arithmetic, boolean, strings)
  - Functions/Procedures
  - Commands
- We will examine carefully how these are defined syntactically, used semantically, and implemented pragmatically.
- We will develop an understanding of different strategies for expression and the impact on language design.
- We will see how types allow the programmer to specify *constraints* on functions and data, and understand the trade-offs of static versus dynamic typing.

Unifying Concepts

- Elements of the following languages will be studied to enhance your practical understanding of how concepts map to features in modern languages
  - C for efficient imperative programming with static types.
  - C++ for object-oriented programming with static types and ad hoc, subtype and parametric polymorphism.
  - Java for imperative, object-oriented, and concurrent programming with static typing and garbage collection.
  - Scheme for lexically scoped applicative-style recursive programming with latent or dynamic typing.
  - Standard ML for practical functional programming with *strict (eager) evaluation* & *polymorphic type inferencing*.
  - Haskell for pure functional programming with *non-strict (lazy) evaluation*.

Programmer Productivity

- Software engineers like to think about programmer productivity
  - how many Lines of Code (LOC), and number of bugs per line of code, etc.
- A programming language can either encourage or discourage the introduction of errors.
  - from a software engineering perspective, we want to build error-free programs.
  - this means the language needs to offer *linguistic mechanisms* that facilitate the production of reliable software.
    - E.g., modularity, strong typing, assertions, exceptions

Programming with Types

- Type definition mechanisms allowing the specification of clearly defined intent through the use of type declarations for both functions/procedures and data.
    - type declarations and strongly typed function *signatures*
    - mechanisms for defining independent lexical scopes to provide for modularization of a program's procedures and data.
  - The ability to define libraries of subprograms, encapsulate data into Abstract Data Types (ADTs)
    - define interfaces as abstraction boundaries for the purpose of information hiding and localized access to data (i.e., no global data) are all critical elements of cleanly organizing a program.
  - The ability to modularly organized data types and/or functions/procedures using modules or classes and to share, reuse or extend a program
  - Generally results in "safer" programs as errors are often detected at compile-time instead of run-time

Static vs Dynamic Typing

- Most compiled languages are statically typed while interpreted languages are often dynamically typed.
  - **Static typing** is done at compile-time
    - generally deemed "safer" since type errors can be detected early, at compile-time, before a program is executed.
    - static typing means that the type of a variable is determined at compile-time and that type constrains the set of values that the variable can hold at run-time.
  - **Dynamic typing** is done at run-time
    - so programmer type errors are not detected until the program is executed.
    - this implies that a type error in an infrequently executed part of a program may not be discovered for a long time, and only surfaces in an unusual (but critical) situation, which then causes the program to fail.
    - dynamic typing means that types are associated with values at run-time
      - the type of a variable may change dynamically during execution to conform to the type of the value that is currently being referenced by the variable.

Program Correctness

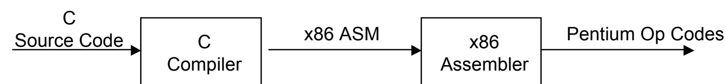
- The ability to assert formal correctness statements about critical parts of a program and reason effectively.
    - A program is intended to carry out a specific computation, but a programmer can fail to adequately address all data value ranges, input conditions, system resource constraints, memory limitations, etc.
    - For mission critical applications, the programmer needs to be able to reason effectively about the behavior of a program.
    - This means that the features of the language 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.

Language Translation

- There are three basic approaches to translation
  - Translation to machine code using a **native-code compiler**
    - followed by execution on a real machine
  - Translation into an internal form and immediate execution via a **read-eval-print-loop interperter**
  - Translation to byte code using a **byte-code compiler**
    - followed by execution on a **virtual machine** (e.g., JVM)
  - There are also hybrid approaches.
    - Source-to-source translation
      - E.g., C++ -> C -> ASM -> machine code
    - new types of byte-code compilers have appeared in recent years, which are called Just-In-Time (JIT) compilers.
      - JIT compilers translate compiled bytecodes into native machine code the first time the bytecode is executed by the virtual machine.
      - Subsequent requests to execute the same bytecode sequence result in the native code being executed, resulting in a significant speedup in execution.

Language Compilation

- A compiler is a program that translates a **source language** into a **target language**
  - the target language is often, but not always, the assembly language for a particular machine
    - the assembly language is then assembled by an assembler into machine code (e.g., x86) for a specific machine



Language Compilation

- The purpose of the compiler is to translate the **syntactic constructs** of the source language into a set of machine instructions that when executed will correctly carry out the computations specified by the source language program.
  - Does lexical analysis, syntax checking, type checking, optimization, and code generation.
  - Programmer logic errors and automatic or explicit type conversions, resulting in loss of precision, may produce incorrect results.
    - *Assigning a float to an int may result in truncation of required arithmetic precision*
    - Subtle logic errors:
      - if (a = b) …. when if (a == b) was intended

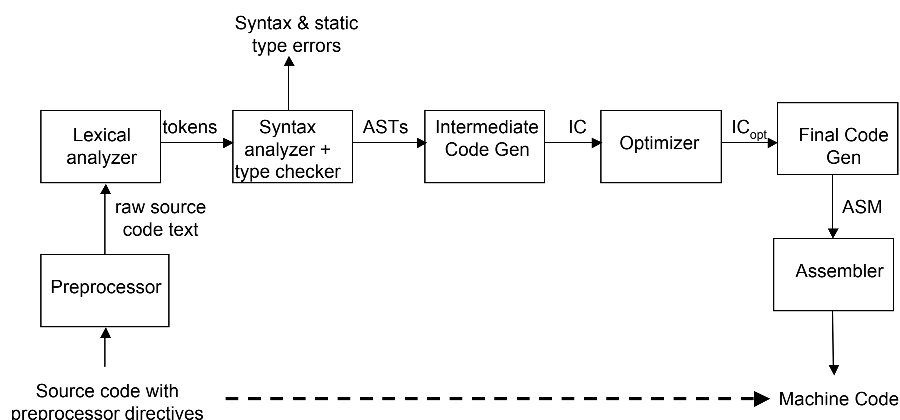
Language Compilation

- In a compiled language, the source program is analyzed for both syntactic correctness and type correctness at compile-time.
  - type errors are identified and reported by static (compile-time) analysis of the source program.
  - note that almost all compiled languages are statically typed (*e.g., C, C++, C#, Java*). However, there are statically typed languages that are interpreted rather than compiled (*e.g., ML*).
- Static determination of all type information in a source program leads to "safer" and usually more efficient run-time code.
  - a side-effect is that all static type information is stripped away at compile-time.

Compilation Phases

- Compilation typically consists of several distinct translation phases. Each phase performs a well-defined transformation on its input.
  - **Preprocessing** does conditional macro text substitution
  - **Lexical analysis** reduces the source program to a sequence of tokens, eliminating comments and extra whitespace characters such as double spaces, tab, and newline.
    - tokens are a more compact representation for the keywords, identifiers, and constant values used in the source program
  - **Syntactic analysis** verifies that collections of tokens output by the lexical analyzer form correct syntactic constructs
    - generates Abstract Syntax Trees (AST) representing the syntactic structure of the program. Static type checking is also done at this phase
  - **Intermediate code generation** "walks" the ASTs and generates intermediate code
  - **Optimizations** may be applied to the intermediate code to produce a more space and/or time efficient program
  - **Final Code generation** & **Assembly** results in target machine specific code optimizations and generation of machine code

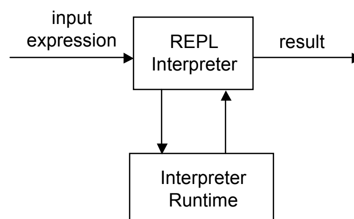
Compilation Phases



Language Interpretation

- An alternative to compilation is interpretation.
  - interpreters are usually implemented by a read-eval-print-loop, which repeatedly reads expressions, evaluates them, and prints the result of the evaluation.
    - reads an input expressions of the source language, usually translating it to some internal form
    - evaluates the internal form of the expression
    - prints the result of the evaluation
    - loops and reads the next input expression until exit
  - interpreters implement some type of abstract machine on top of a real machine.
    - an interpreter usually has a core component, called the interpreter "run-time" that is a compiled (usually imperative) program that runs on the native machine.
    - the interpreter together with its run-time implements an abstract machine which is capable of executing expressions written in an interpreted language.

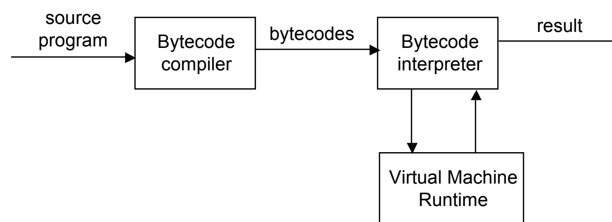
Read-Eval-Print-Loop Interpreter



Bytecode Compilation

- Interpreted languages are often criticized as being too inefficient for practical applications because of the read-eval-print-loop mechanism.
  - an alternative is to combine compilation with interpretation so that expressions and statements can be pre-compiled, and then interpreted
  - Bytecodes are conceptually similar to real machine op-codes except they represent compiled instructions to a **virtual machine** instead of a real machine.
  - a bytecode compiler statically compiles a source program into a set of bytecodes for some virtual machine which are then interpreted by a bytecode interpreter that implements the virtual machine.
  - bytecode interpreters are usually much more efficient than plain read-eval-print loop interpreters. To gain efficiency, a JIT compiler may be used to dynamically compile bytecodes into native machine code.
  - **Question: In what way are bytecodes "better" then real machine opcodes?**

Bytecode Compilation & Interpretation



Homework Assignment #1
Due Monday, Sept 10

- Implement the factorial function shown earlier using C, C++ or Java. Execute the factorial function for successive values of *n, e.g,. n=0,1,2,…*
  - Turn in a printed program listing showing the source code and the output. Report the highest value of *n* for which you are able to correctly compute n!
- Using the DrScheme system implement the factorial function in Scheme as shown before
  - Turn in a printout of your program and a trace of computing (fact n) for n = 0,1,2,…
    - Use the print function in DrScheme to print both the "definitions window" and the "evaluation window"
  - what is the maximum value for n for which you can correctly computer n! ?

Homework Assignment

- The factorial function is commonly defined as a *recursive* function for the sake of elegance, even though it is more space efficient to implement it iteratively. The recursive factorial function is *linear recursive* since there is only one recursive call in the body. A sophisticated language compiler/interpreter can sometimes optimize a linear recursive function by automatically turning it into an equivalent iteration if the function is written such that it is **tail recursive**.

Homework Assignment

- The optimization is possible because a tail recursive function does not require stack space for maintaining a return context for the recursion in order to complete a computation before returning a final result.
  - For example, the computation *n * fact(n-1)* requires the recursive call to *fact* to return into the calling context so that *n* can be multiplied against the result of *fact(n-1).*
  - If this return to do the multiplication after the recursion could be eliminated, no extra stack frame would be required for each successive function invocation.
- Re-implement the factorial function in C or C++ so that it is tail recursive.
  - Hint #1: make changes so that it is not necessary for the recursive function call to return into a recursive calling context to do the multiplication.
  - Hint #2: implement an iterative version of the factorial function first, and then consider how to convert it to a tail recursive factorial implementation.

Homework Assignment

- DrScheme
  - DrScheme is installed on CS Linux machines in
    - /lusr/bin/drscheme
  - You can also download it from the following website for Windows, Linux or Mac OS X:
    - http://www.drscheme.org

Practical Programming Issues

- C/C++ Compilers:
  - GNU gcc and g++ are installed on departmental Linux and Solaris systems:
    - `unix% /lusr/gnu/bin/gcc`
      `unix% /lusr/gnu/bin/g++`
- Java
  - On CS Linux and Solaris machines you can run the Java compiler (javac) and interpreter using the commands:
    - `unix% /lusr/java/bin/javac Foo.java`
    - `unix% /lusr/java/bin/java Foo`
  - The Java 2 Platform (J2SE 1.4.x or later) is also available direct from:
    - http://java.sun.com