CMSC 330: Organization of Programming Languages

Introduction

Course Goal

Learn how programming languages "work"

- Broaden your language horizons
  - Different programming languages
  - Different language features and tradeoffs
- Study how languages are implemented
  - What *really* happens when I write x.f(…)?
- Study how languages are described

Other goals

- Learn some fundamental CS concepts
  - Regular expressions
  - Context free grammars
  - Automata theory
  - Compilers & parsing
  - Parallelism & synchronization
- Improve programming skills
  - Learn how to learn new programming languages
  - Learn how to program in a new programming style

All Languages Are Equivalent

- A language is *Turing complete* if it can compute any function computable by a Turing Machine
- Essentially all general-purpose programming languages are Turing complete
- Therefore this course is useless!

Why Study Programming Languages?

- Using an appropriate language for a problem may be easier, faster, and less error-prone
- To make you better at learning new languages
  - You may need to add code to a legacy system
    - E.g., FORTRAN, COBOL,…
  - You may need to write code in a new language
    - Your boss says, “From now on, all software will be written in Ada/C++/Java/…”

Why Study Programming Languages?

- To make you better at using languages you already know
  - Many “design patterns” in Java are functional programming techniques

Changing Language Goals

- 1950s-60s – Compile programs to execute efficiently
  - Language features based on hardware concepts
    - Integers, reals, goto statements
  - Programmers cheap; machines expensive
    - Keep the machine busy

Changing Language Goals

- Today
  - Language features based on design concepts
    - Encapsulation, records, inheritance, functionality, assertions
  - Processing power and memory very cheap; programmers expensive
    - Ease the programming process

Language Attributes to Consider

- Syntax
  - What a program looks like
- Semantics
  - What a program means
- Implementation
  - How a program executes

Imperative Languages

- Also called *procedural* or *von Neumann*
- Building blocks are functions and statements
- Programs that write to memory are the norm

```
int x = 0;
while (x < y) x := x + 1;
```

  - FORTRAN (1954)
  - Pascal (1970)
  - C (1971)

Functional Languages

- Also called *applicative* languages
- No or few writes to memory
- Functions are higher-order

```
let rec map f = function [] -> []
           | x::l -> (f x)::(map f l)
```

  - LISP (1958)
  - ML (1973)
  - Scheme (1975)
  - Haskell (1987)

Logical Languages

- Also called *rule-based* or *constraint-based*
- Program consists of a set of rules
  - "A :- B" – if B holds, then A holds
    - `append([], L2, L2).`
    - `append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).`
  - PROLOG (1970)
  - Various expert systems

Object-Oriented Languages

- Programs are built from objects
- Objects combine functions and data
- Often have classes and inheritence
- "Base" may be either imperative or functional

```
class C { int x; int getX() {return x;} … }
class D extends C { … }
```

  - Smalltalk (1969)
  - C++ (1986)
  - Java (1995)

Scripting Languages

- Rapid prototyping languages ideal for "little" tasks
- Typically with rich text processing abilities
- Generally very easy to use
- "Base" may be imperative or functional; may be OO

```
#!/usr/bin/perl
for ($j = 0; $j < 2 * $lc; $j++) {
     $a = int(rand($lc));
…
```

  - sh (1971)
  - perl (1987)
  - Python (1991)
  - Ruby (1993)

"Other" Languages

- There are lots of other languages around with various features
  - COBOL (1959) – business applications
  - BASIC (1964) – MS Visual Basic widely used
  - Logo (1968) – introduction to programming
  - Forth (1969) – Mac Open Firmware
  - Ada (1979) – the DoD language
  - Postscript (1982) – printers
  - …

Languages You Know

- So far at UMD, you've seen two main languages
  - Java – object-oriented imperative language
  - C – imperative language without objects
- This course: two new languages
  - Plus we'll see snippets of other languages

Ruby

- An imperative, object-oriented scripting language
  - Created in 1993 by Yukihiro Matsumoto
  - Similar in flavor to many other scripting languages (e.g., perl, python)
  - Much cleaner than perl
  - Full object-orientation (even primitives are objects!)

A Small Ruby Example

intro.rb:

```
def greet(s)
  print("Hello, ")
  print(s)
  print("!\n")
end
```

```
% irb     # you'll usually use "ruby" instead
irb(main):001:0> require "intro.rb"
=> true
irb(main):002:0> greet("world")
Hello, world!
=> nil
```

OCaml

- A mostly-functional language
  - Has objects, but won't discuss (much)
  - Developed in 1987 at INRIA in France
  - Dialect of ML (1973)
- Natural support for pattern matching
  - Makes writing certain programs very elegant
- Has a really nice module system
  - Much richer than interfaces in Java or headers in C
- Includes type inference
  - Types checked at compile time, but no annotations

A Small OCaml Example

intro.ml:

```
let greet s =
  begin
    print_string "Hello, ";
    print_string s;
    print_string "!\n"
  end
```

```
$ ocaml
        Objective Caml version 3.08.3

# #use "intro.ml";;
val greet : string -> unit = <fun>
# greet "world";;
Hello, world!
- : unit = ()
```

Attributes of a Good Language

1. Clarity, simplicity, and unity
   - Provides both a framework for thinking about algorithms and a means of expressing those algorithms
2. Orthogonality
   - Every combination of features is meaningful
   - Features work independently
     - What if, instead of working independently, adjusting the volume on your radio also changed the station? You would have to carefully change both simultaneously and it would become difficult to find the right station and keep it at the right volume.

Attributes of a Good Language

3. Naturalness for the application
   - Program structure reflects the logical structure of algorithm
4. Support for abstraction
   - Program data reflects problem being solved
5. Ease of program verification
   - Verifying that program correctly performs its required function

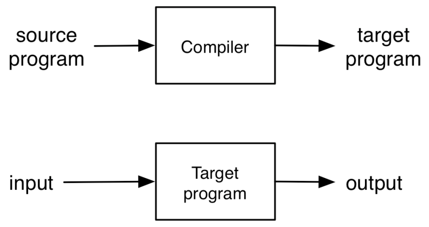
Attributes of a Good Language

6. Programming environment
   - External support for the language
7. Portability of programs
   - Can develop programs on one computer system and run it on a different computer system
8. Cost of use
   - Program execution (run time), program translation, program creation, and program maintenance
9. Security & safety
   - Should be very hard to write unsafe program

Executing Languages

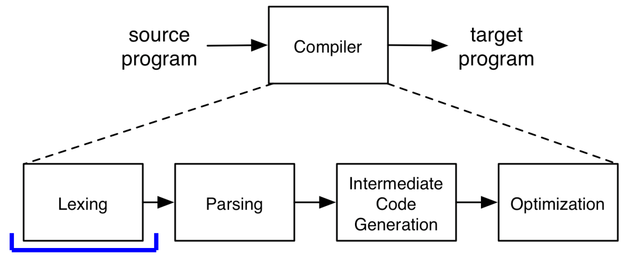
- Consider a high-level language (i.e., not machine code). There are two main ways which the language can use for executing programs: compilation versus interpretation

Compilation or Translation



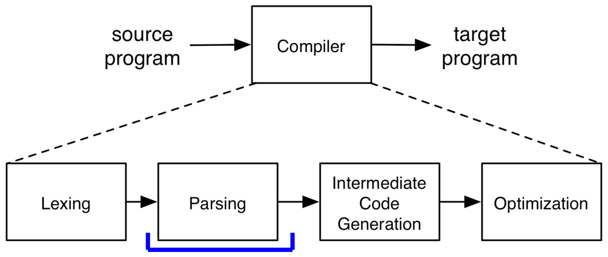
- Source program translated to another language
  - Often machine code, which can be directly executed

Steps of Compilation



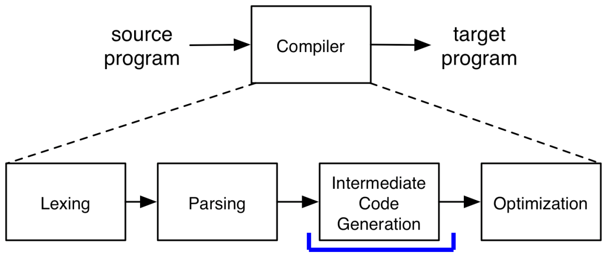
1. Lexical analysis (scanning) – break up source code into *tokens* such as numbers, identifiers, keywords, and operators

Steps of Compilation



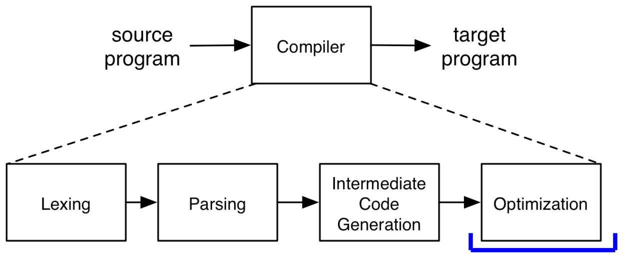
2. Parsing (syntax analysis) – group tokens together into higher-level language constructs (conditionals, assignment statements, functions, …)

Steps of Compilation



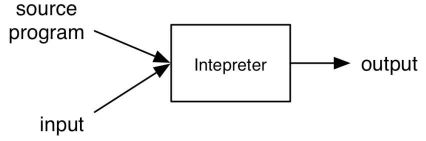
3. Intermediate code generation – verify that the source program is valid and translate it into an internal representation
   - May have more than one intermediate representation

Steps of Compilation



4. Optimization (optional) – improve the efficiency of the generated code
   - Eliminate dead code, redundant code, etc.

Interpretation



- Interpreter executes each instruction in source program one step at a time
  - No separate executable
- Advantages and disadvantages of compilation versus interpretation?

Compiler or Intepreter?

gcc
- Compiler – C code translated to object code, executed directly on hardware

javac
- Compiler – Java source code translated to Java byte code

tcsh/bash
- Interpreter – commands executed by shell program

java
- Interpreter – Java byte code executed by virtual machine