LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.

HUH?

SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.

AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.

THE PATTERNS AND METAPATTERNS DANCED.

SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.

TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.

NO, IT'S NOT.

IT'S NOT?

I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.

I WONDER IF THE CYCLES WILL CONTINUE FOREVER.

A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.

THESE ARE YOUR FATHER'S PARENTHESES

ELEGANT WEAPONS FOR A MORE... CIVILIZED AGE.
CMSC 330: Organization of Programming Languages

Introduction

Instructor: Michael Lam (lam at cs)
TA: Aleks Aris (aris at cs)
Course Website


- Syllabus
- Office hours
- Schedule
- Project assignments
- Resource links
- Web forum
- Submit server
Course Grading

• 5 projects (40%)
  – Late submission penalty (see syllabus)
• 2 midterms + final exam (15/15/25%)
• Class participation (5%)

• Homeworks will not be graded and solutions will be posted online
Classroom reminders

• Arrive on time
• Sit up front (easier to see console text)
• Put cell phones on silenced/vibrate
• Laptops are fine, but be polite
• Interaction is good—ask questions!
• Attendance is neither mandatory nor directly rewarded
  – Lots of in-class examples
• Lectures will be available online
Project rules

• DO NOT SHARE PROJECT CODE!
• Re-using your own code is fine
• We will be using the submit server
  – Submit EARLY to ensure your setup works
• Most tests will be secret
  – Spend time writing your own test cases
  – Sharing test cases is fine
• Secret test results available after Friday discussion section
• Project late penalties
  – 10% up to 24 hours late
  – 25% between 24 hours and 9am Friday
Other notes

• Take advantage of the web forum
• No textbook - use the lecture notes
• The summer semester is highly compressed
  – Stay organized and ahead of your work
• Get help as soon as you need it
  – Office hours are posted online
• GLUE/GRACE accounts
  – glue.umd.edu/grace.umd.edu
  – UM directory ID+password
  – If you have a laptop, use it instead
Syllabus

• Scripting languages (Ruby)
• Regular expressions and finite automata
• Context-free grammars
• Functional programming (OCaml)
• Concurrency
• Object-oriented programming (Java)
• Environments, scoping, and binding
• Advanced topics
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.foo(…)?

• Study how languages are described
  – Mathematical formalisms
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

• So why is this course not useless?
Why Study Programming Languages?

• Using the right language for a problem may be easier, faster, and less error-prone
  – Programming is a human activity
  – Features of a language make it easier or harder to program for a specific application
• To make you better at learning new languages
  – You may need to add code to a legacy system
    • E.g., FORTRAN (1954), COBOL (1959), …
  – 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/…}”

• You may think Java is the ultimate language, but if you are still programming or managing programmers in 20 years, they probably won’t be programming in Java!
Why Study Programming Languages?

• To make you better at using languages you think you already know
  – Many “design patterns” in Java are functional programming techniques
  – Understanding what a language is good for will help you know when it is appropriate to use
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

• 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
Types of Languages

- Imperative/Procedural
- Declarative
  - Functional
  - Logical (rule-based)
- Object-oriented
- Scripting
- Data-oriented/Query (not covered)
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)
- OCaml (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 inheritance
  – “Base” may be either imperative or functional
    ```
    class C { int x; int getX() {return x;} ... }
    class D extends C { ... }
    ```

  – Smalltalk (1969)
  – C++ (1986)
  – OCaml (1987)
Scripting Languages

• Rapid prototyping languages for “little” tasks
  – Typically with rich text processing abilities
  – Generally very easy to use
  – “Base” may be imperative or functional; may be OO

```perl
#!/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
    - Imperative, rich file structure
  - BASIC (1964) – MS Visual Basic widely used
    - Originally an extremely simple language
    - Now a single word oxymoron
  - Logo (1968) – Introduction to programming
  - Forth (1969) – Mac Open Firmware
    - Extremely simple stack-based language for PDP-8
  - Ada (1979) – The DoD language
    - Realtime
  - Postscript (1982) – Printers- Based on Forth
- ...
Executing Languages

• Suppose we have a program $P$ written in a high-level language (i.e., not machine code)

• There are two main ways to run $P$
  – Compilation
  – Interpretation
Compilation or Translation

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

```
def greet(s):
    print("Hello, ")
    print(s)
    print("!
")
end
```
Steps of Compilation

1. Lexical Analysis (Scanning) – Break up source code into *tokens* such as numbers, identifiers, keywords, and operators

```plaintext
FOR
ID:i
LESSTHAN
NUM:6
```
Steps of Compilation

2. Parsing (Syntax Analysis) – Group tokens together into higher-level language constructs (conditionals, assignment statements, functions, …)
3. Intermediate Code Generation – Verify that the source program is valid and translate it into an internal representation
   – May have more than one intermediate rep

for 0:i:6
or
Load 0
Load i
Etc…
4. Optimization (optional) – Improve the efficiency of the generated code
   - Eliminate dead code, redundant code, etc.
   - Change algorithm without changing functionality
     (e.g., $X=Y+Y+Y+Y \Rightarrow X=4\times Y \Rightarrow X = Y$ shift left 2)

[If interested in compilation, take CMSC 430]
Interpretation

- Interpreter executes each instruction in source program one step at a time
  - No separate executable
  - Advantages? Disadvantages?

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

"world"
Compiler or Interpreter?

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
Compilation or Interpretation – Not so simple today

• Previously
  – Build program to use hardware efficiently
  – Often use of machine language for efficiency

• Today
  – No longer write directly in machine language
  – Use of layers of software
  – Concept of virtual machines
    • Each layer is a machine that provides functions for the next layer (e.g., javac/java distinction)
    • This is an example of *abstraction*, a basic building-block in computer science
Interactive Exercise

Introduce yourself to your neighbor(s) in groups of 2-4 and together write down as many languages as you can think of, specifying the following attributes:

Type: imperative, functional, or logical

Properties: object-oriented, scripting, or query

Execution: compiled, interpreted, or both
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
   - 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. Your radio and tuning work orthogonally. And aren’t you glad they do!
Attributes of a Good Language

1. Naturalness for the application
   • Program structure reflects the logical structure of algorithm

2. Support for abstraction
   • Program data reflects problem being solved

3. Ease of program verification
   • Verifying that program correctly performs its required function
Attributes of a Good Language

1. Programming environment
   • External support for the language

2. Portability of programs
   • Transportability of the resulting programs from the computer on which they are developed to other computer systems

3. Cost of use
   • Program execution, program translation, program creation, and program maintenance
Who defines a language?

Is: \( I = 1 \land \land 2 + 3 \mid 4; \) legal in C?
  - What is assigned to \( I \) if it is?
  - Who makes this determination?

3 ways typically to answer this:
  1. Read language manual (Problem: Can you find one?)
  2. Read language standard (Problem: Have you ever seen it?)
  3. Write a program to see what happens. (Easy to do!)

Most programmers do 3
**Creation of standards**

Language standards defined by national standards bodies:
- ISO - International Standards organization
- IEEE - Institute of Electrical and Electronics Engineers
- ANSI - American National Standards Institute

All work in a similar way:
1. Working group of volunteers set up to define standard
2. Agree on features for new standard
3. Vote on standard
4. If approved by working group, submitted to parent organization for approval
Creation of standards

- Standards in the US are voluntary:
  - There is no federal standards-making organization.
  - NIST - National Institute for Standards and Technology develops standards that are only required on federal agencies, not for commercial organizations.

- Consensus is the key to standards making:
  - Contentious features often omitted to gain consensus
  - Only vendors have a vested interest in the results
  - Users don't care until standard approved, and then it is too late!
Standards conforming programs

• Standards define behavior for a *standards conforming* program - one that meets the rules of the language standard

• In general (except for Ada), behavior of non-conforming program is not specified, so any extensions to a standards conforming compiler may still be standards conforming, even though the program is not standards conforming.

• Standards supposed to be reviewed every 5 years
  – Examples:
    – Ada 1983, 1995

• Not quite 5 years, but at least periodically
When to standardize a language?

- Problem: When to standardize a language?
  - If too late - many incompatible versions
    - FORTRAN in 1960s was already a de facto standard, but no two were the same
    - LISP in 1994, about 35 years after developed.
  - If too early - no experience with language - Ada in 1983 had no running compilers
  - Just right - Probably Pascal in 1983, although it is now mostly a dead language

- Other languages:
  - C in 1988
  - De facto standards: ML, SML, OCaml, Ruby
  - Smalltalk - none
  - Prolog - none
Internationalization

• Programming has become international
  – I18N issue - Internationalization - How to specify languages useful in a global economy?

• Character sets:
  – 1950s-1960s – 6 bit sufficient (upper case, digits, special symbols …)
  – ASCII is a 7 bit 128 character code
  – Single 8-bit byte; usual format today - 256 character values. A lot in 1963, but insufficient today

• What about other languages?
  – Additional letters: German umlaut-ä, French accent-é, Scandinavian symbols-ö,
  – Russian, other alphabets (Greek, Arabic, Hebrew), ideographs (Chinese, Korean)?
  – Unicode - 16 bit code allows for 65K symbols. 8-bit byte is insufficient
Internationalization

• Some of the internationalization issues:
  – What character codes to use?
  – Collating sequences? - How do you alphabetize various languages?
  – Dates? – If I said your exam was on 10/12/07 when would you show up?
• Time? - How do you handle time zones, summer time in Europe, daylight savings time in US, Southern hemisphere is 6 months out of phase with northern hemisphere, Date to change from summer to standard time is not consistent. Some zones 30 minutes off.
• Currency? - How to handle dollars, pounds, euros, etc.
Summary

• Reasons to study PL
• Types of languages:
  – Imperative
  – Functional
  – Logical
  – Object-oriented
• Execution methods:
  – Compilation
  – Interpretation
• Desirable language properties
• Standardization
Ruby

• An imperative, object-oriented scripting language
  – Created in mid-1990s by Yukihiro Matsumoto
  – Similar in flavor to many other scripting languages (e.g., perl, python)
  – Much cleaner than perl
  – Fully object-oriented (even primitives are objects!)
A Small Ruby Example

```
intro.rb:
def greet(s)
    print("Hello, ")
    print(s)
    print("!\n")
end
```

```ruby
% 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

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

$ ocaml
    Objective Caml version 3.08.3

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

• Acquaint yourself with the website
• Review today's slides
• Check GLUE/GRACE and forum accounts
• Install Ruby (and OCaml if you have time)