G22.2110-001 Programming Languages - Fall 2018

Class 1

Thomas Wies

New York University

Course Information and Resources

Course web page (general info, syllabus, etc.)
http://cs.nyu.edu/wies/teaching/pl-fa18/

Piazza (announcements, course related discussions)
https://piazza.com/class/jbcu67g5izt5kr

You should already be enrolled. Please complete the questionnaire!

Github (class notes and code)
https://github.com/nyu-pl-fa18

NYU classes (grade distribution only)

Important Dates

Class Meetings

Wed 5:10-7pm in CIWW 109

Office hours: Thomas Wies, Tue 4-5pm

Recitations

Thu 5:10-7pm in CIWW 109

Office hours: Zvonimir Pavlinovic, Fri 10:30am-12pm

Midterm Exam

Wed Oct 24, 5:10-6:10pm in CIWW 109

Final Exam

Wed Dec 19, 5:10-7pm in CIWW 109

Grading

Grading Key

- 30% for homework assignments
- 30% for midterm exam
- 40% for final exam

Submission Policies

- All work must be your own.
- Solutions must be submitted before the announced date and time deadline for full credit.
- For every 24 hours late you lose 10%.
- Late solutions will not be accepted after the late deadline. (usually one week).
- If you turn in a solution that does not compile, it will not be accepted. You can resubmit according to the above rules.

Textbooks

- No required textbooks!
- Though, course web page lists several good books that I recommend.
- Also, see the syllabus for pointers for recommended reading.

What is a Program?

Programs

What is a Program?

- A fulfillment of a customer's requirements
- An idea in your head
- A sequence of characters
- A mathematical interpretation of a sequence of characters
- A file on a disk
- Ones and zeroes
- Electromagnetic states of a machine

Programs

What is a Program?

- A fulfillment of a customer's requirements
- An idea in your head
- A sequence of characters
- A mathematical interpretation of a sequence of characters
- A file on a disk
- Ones and zeroes
- Electromagnetic states of a machine

In this class, we will typically take the view that a program is a sequence of characters, respectively, its mathematical interpretation.

Though we recognize the close association to the other possible meanings.

Programs

What is a Program?

- A fulfillment of a customer's requirements
- An idea in your head
- A sequence of characters
- A mathematical interpretation of a sequence of characters
- A file on a disk
- Ones and zeroes
- Electromagnetic states of a machine

In this class, we will typically take the view that a program is a sequence of characters, respectively, its mathematical interpretation.

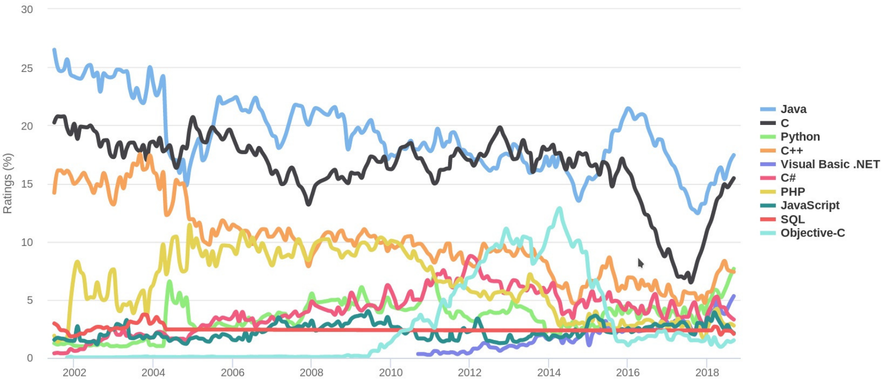
Though we recognize the close association to the other possible meanings.

A *Programming Language* describes what sequences are allowed (the syntax) and what they mean (the semantics).

Why Study Programming Languages?

Why Study Programming Languages?

TIOBE Programming Language Popularity Index



Why Study Programming Languages?

1. *Understand foundations*: understanding notation and terminology helps to learn complicated features and new languages quickly
2. *Make good choices when writing code*: understanding benefits and costs helps you make good choices when programming
3. *Better debugging and profiling*: sometimes you need to understand what's going on under the hood
4. *Simulate useful features in languages that lack them*: being exposed to different idioms and paradigms broadens your set of tools and techniques
5. *Make better use of language technology*: parsers, analyzers, optimizers appear in many contexts
6. *Leverage extension languages*: many tools are customizable via specialized languages (e.g. emacs elisp)

A Brief History of Programming Languages

The first computer programs were written in *machine language*.

Machine language is just a sequence of ones and zeroes.

The computer interprets sequences of ones and zeroes as *instructions* that control the *central processing unit* (CPU) of the computer. The length and meaning of the sequences depends on the CPU.

Example

> *On the 6502, an 8-bit microprocessor used in the Apple II computer, the following bits add 1 and 1:* 10101001000000010110100100000001*.*
>
> *Or, using base 16, a common shorthand:* A9016901*.*

Programming in machine language requires an extensive understanding of the low-level details of the computer and is extremely tedious if you want to do anything non-trivial.

But it *is* the most straightforward way to give instructions to the computer: no extra work is required before the computer can run the program.

A Brief History of Programming Languages

Before long, programmers started looking for ways to make their job easier. The first step was *assembly language*.

Assembly language assigns meaningful names to the sequences of bits that make up instructions for the CPU.

A program called an *assembler* is used to translate assembly language into machine language.

Example

The assembly code for the previous example is:

```
LDA #$01
ADC #$01
```

Question: *How do you write an assembler?*

A Brief History of Programming Languages

Before long, programmers started looking for ways to make their job easier. The first step was *assembly language*.

Assembly language assigns meaningful names to the sequences of bits that make up instructions for the CPU.

A program called an *assembler* is used to translate assembly language into machine language.

Example

The assembly code for the previous example is:

```
LDA #$01
ADC #$01
```

Question: *How do you write an assembler?*

Answer: in machine language (at least the first time)!

A Brief History of Programming Languages

As computers became more powerful and software more ambitious, programmers needed more efficient ways to write programs.

This led to the development of *high-level* languages, the first being Fortran.

High-level languages have features designed to make things much easier for the programmer.

In addition, they are largely *machine-independent*: the same program can be run on different machines without rewriting it.

But high-level languages require a *compiler*. The compiler's job is to convert high-level programs into machine language. Alternatively, the program can be interpreted by another program, an *interpreter*, which is already compiled to machine language. More on this later...

Question: *How do you write a compiler?*

A Brief History of Programming Languages

As computers became more powerful and software more ambitious, programmers needed more efficient ways to write programs.

This led to the development of *high-level* languages, the first being Fortran.

High-level languages have features designed to make things much easier for the programmer.

In addition, they are largely *machine-independent*: the same program can be run on different machines without rewriting it.

But high-level languages require a *compiler*. The compiler's job is to convert high-level programs into machine language. Alternatively, the program can be interpreted by another program, an *interpreter*, which is already compiled to machine language. More on this later...

Question: *How do you write a compiler?*

Answer: in assembly language (at least the first time)

Programming Languages

There are now thousands of programming languages.
Why are there so many?

Programming Languages

There are now thousands of programming languages.
Why are there so many?

- *Evolution*: old languages evolve into new ones as we discover better and easier ways to do things:
  Algol $\Rightarrow$ Bcpl $\Rightarrow$ C $\Rightarrow$ C++ $\Rightarrow$ Java $\Rightarrow$ Scala
- *Special Purposes*: some languages are designed specifically to make a particular task easier. For example, SML was originally designed to write proof tactics for a theorem prover.
- *Personal Preference*: Programmers are opinionated and creative. If you don't like any existing programming language, why not create your own?

Programming Languages

Though there are many languages, only a few are widely used.
What makes a language successful?

Programming Languages

- *Expressive Power*: Though most programming languages share the same theoretical power (i.e. they are *Turing complete*), it is often much easier to accomplish a task in one language as opposed to another.
- *Ease of Use for Novices*: Python and Java owe much their popularity to the fact that they are easy to learn.
- *Ease of Implementation*: Languages like Lisp, Pascal, Java were designed to be easily portable from one platform to another.
- *Standardization*: The weak standard for Pascal is one important reason it declined in favor in the 1980's.
- *Open Source*: C owes much of its popularity to being closely associated with open source projects.
- *Excellent Compilers*: A lot of resources were invested into writing good compilers for Fortran, one reason why it is still used today for many scientific applications.
- *Economics, Patronage, Inertia*: Some languages are supported by large and powerful organizations: Ada by the Department of Defense; Java by Oracle; C# by Microsoft. Some languages live on because of a large amount of legacy code.

Language Design

Language design is influenced by various viewpoints

- *Programmers*: Desire expressive features, predictable performance, supportive development environment
- *Implementors*: Prefer languages to be simple and semantics to be precise
- *Verifiers/testers*: Languages should have rigorous semantics and discourage unsafe constructs

The interplay of design and implementation is important to understand limitations and idiosyncrasies of programming language features and we will return to this theme periodically.

Classifying Programming Languages

Programming Paradigms

- *Imperative (von Neumann)*: Fortran, Pascal, C, Ada
  - programs have mutable storage (state) modified by assignments
  - the most common and familiar paradigm
- *Object-Oriented*: Simula 67, Smalltalk, Ada, Java, Python, C#, Scala, JavaScript
  - data structures and their operations are bundled together
  - inheritance, information hiding
- *Functional (applicative)*: Scheme, Lisp, SML, OCaml, Haskell
  - based on lambda calculus
  - functions are first-class objects
  - *side effects* (e.g., assignments) discouraged
- *Logical (declarative)*: Prolog, Mercury
  - programs are sets of assertions and rules

Classifying Programming Languages

Multi-Paradigm Languages

Many modern general-purpose languages support multiple paradigms.

- *Imperative + Object-Oriented*: C++, Java, C#, Objective-C
- *Functional + Object-Oriented*: OCaml, F#, Scala
- *Functional + Logical*: Curry, Oz

Concurrent Programming

- Not really a category of programming languages
- Usually implemented with extensions within existing languages
  - Threads (e.g. Pthreads in C)
  - Actors (e.g. Erlang and Scala)
  - Futures and promises
  - ...
- Some exceptions (e.g. *dataflow* languages)

Classifying Programming Languages

Compared to machine or assembly language, all others are high-level.

But within high-level languages, there are different levels as well.

Somewhat confusingly, these are also referred to as low-level and high-level.

- *Low-level* languages give the programmer more control (at the cost of requiring more effort) over how the program is translated into machine code.
  - C, Fortran
- *High-level* languages hide many implementation details, often with some performance cost
  - Basic, Lisp, Scheme, SML, Prolog, Haskell, Python
- *Wide-spectrum* languages try to do both:
  - Ada, C++, Rust, (Java)
- High-level languages typically have garbage collection and are often interpreted.
- The higher the level, the harder it is to predict performance (bad for real-time or performance-critical applications)

Programming Idioms

- All general-purpose languages have essentially the same capabilities (they are Turing-complete)
- But different languages can make the same task difficult or easy
  - Try multiplying two Roman numerals
- Idioms in language A may be useful inspiration when writing in language B.

Programming Idioms

- Copying a string q to p in C:

```
while (*p++ = *q++) ;
```

- Removing duplicates from the list @xs in Perl:

```
my %seen = ();
@xs = grep { ! $seen{$_}++; } @xs;
```

- Computing the sum of numbers in list xs in OCaml:

```
List.fold_left (fun sum x -> sum + x) 0 xs
```

  or even shorter

```
List.fold_left (+) 0 xs
```

- Split a list L into all possible consecutive sublists L1 and L2 in Prolog:

```
append(L1, L2, L).
```

Some of these may seem natural to you; others may seem counterintuitive.

One goal of this class is for you to become comfortable with many different idioms.

Characteristics of Modern Languages

Modern general-purpose languages (e.g., Ada, C++, Java) have similar characteristics:

- large number of features (grammar with several hundred productions, 500 page reference manuals, ...)
- a complex type system
- procedural mechanisms
- object-oriented facilities
- abstraction mechanisms, with information hiding
- several storage-allocation mechanisms
- facilities for concurrent programming
- facilities for generic programming
- development support including IDEs, libraries, compilers, build systems

We will discuss many of these in detail this semester.

Compilation vs Interpretation

Compilation

- Translates a program into machine code
  (or something close to it, e.g. byte code)
- Thorough analysis of the input language
- Nontrivial translation

Interpretation

- Executes a program one statement at a time using a virtual machine
- May employ a simple initial translator (preprocessor)
- Most of the work done by the virtual machine
- Often involves *just-in-time compilation* to machine code at run-time

Compilation overview

Major phases of a compiler:

1. *Lexer*: Text $\longrightarrow$ Tokens
2. *Parser*: Tokens $\longrightarrow$ Parse Tree
3. *Intermediate code generation*: Parse Tree $\longrightarrow$ Intermed. Representation (IR)
4. *Optimization I*: IR $\longrightarrow$ IR
5. *Target code generation*: IR $\longrightarrow$ assembly/machine language
6. *Optimization II*: target language $\longrightarrow$ target language

Syntax and Semantics

Syntax refers to the structure of the language, i.e. what sequences of characters are programs.

- Formal specification of syntax requires a set of rules
- These are often specified using *grammars*

Semantics denotes meaning:

- Given a program, what does it mean?
- Meaning may depend on context

We will not be covering semantic analysis (this is covered in the compilers and abstract interpretation courses). Though, I can provide you pointers if you are interested.

We now look at grammars in more detail.

Grammars

A *grammar* G is a tuple (Σ, N, P, S), where:

- N is a set of *non-terminal* symbols
- $S \in N$ is a distinguished non-terminal: the *root* or *start* symbol
- Σ is a set of *terminal* symbols, also called the *alphabet*. We require Σ to be disjoint from N (i.e. $\Sigma \cap N = \emptyset$).
- P is a set of rewrite rules (productions) of the form:

$$\mathrm{ABC} \ldots \rightarrow \mathrm{XYZ} \ldots$$

  where $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}$ are terminals and non-terminals.

Any sequence consisting of terminals and non-terminals is called a *string*.

The *language* defined by a grammar is the set of strings containing *only* terminal symbols that can be generated by applying the rewriting rules starting from S.

Grammars Example

- $N = \{S, X, Y\}$
- $S = S$
- $\Sigma = \{a, b, c\}$
- P consists of the following rules:
  - $S \rightarrow b$
  - $S \rightarrow XbY$
  - $X \rightarrow a$
  - $X \rightarrow aX$
  - $Y \rightarrow c$
  - $Y \rightarrow Yc$

Some sample derivations:

- $S \rightarrow b$
- $S \rightarrow XbY \rightarrow abY \rightarrow abc$
- $S \rightarrow XbY \rightarrow aXbY \rightarrow aaXbY \rightarrow aaabY \rightarrow aaabc$

The Chomsky hierarchy

- Regular grammars (Type 3)
  - All productions have a single non-terminal on the left and a single terminal and optionally a single non-terminal on the right
  - The position of the non-terminal symbol with respect to the terminal symbol on the right hand side of rules must always be the same in a single grammar (i.e. always follows or always precedes)
  - Recognizable by finite state automaton
  - Used in *lexers*
- Context-free grammars (Type 2)
  - All productions have a single non-terminal on the left
  - Right side of productions can be any string
  - Recognizable by non-deterministic pushdown automaton
  - Used in *parsers*

The Chomsky hierarchy

- Context-sensitive grammars (Type 1)
  - Each production is of the form $\alpha A \beta \to \alpha \gamma \beta$,
  - A is a non-terminal, and α, β, γ are arbitrary strings (α and β may be empty, but not γ)
  - Recognizable by linear bounded automaton
- Recursively-enumerable grammars (Type 0)
  - No restrictions
  - Recognizable by turing machine

Regular expressions

An alternate way of describing a regular language over an alphabet Σ is with *regular expressions*.
We say that a regular expression R denotes the language $[\![R]\!]$ (recall that a language is a set of strings).
Regular expressions over alphabet Σ:

- ϵ denotes $\emptyset$
- a character x, where $x \in \Sigma$, denotes $\{x\}$
- (sequencing) a sequence of two regular expressions RS denotes $\{\alpha\beta \mid \alpha \in [\![R]\!], \beta \in [\![S]\!]\}$
- (alternation) $R \mid S$ denotes $[\![R]\!] \cup [\![S]\!]$
- (Kleene star) R^* denotes the set of strings which are concatenations of zero or more strings from $[\![R]\!]$
- parentheses are used for grouping
- $R^? \equiv \epsilon \mid R$
- $R^+ \equiv RR^*$

Regular grammar example

A grammar for floating point numbers:

$$\begin{array}{lcl} \text{Float} & \rightarrow & \text{Digits} \mid \text{Digits . Digits} \\ \text{Digits} & \rightarrow & \text{Digit} \mid \text{Digit Digits} \\ \text{Digit} & \rightarrow & 0|1|2|3|4|5|6|7|8|9 \end{array}$$

A regular expression for floating point numbers:

$$(0|1|2|3|4|5|6|7|8|9)^{+}(.(0|1|2|3|4|5|6|7|8|9)^{+})^{?}$$

The same thing in Perl:

```
[0-9]+(\.[0-9]+)?
```

or

```
\d+(\.\d+)?
```

Tokens

Tokens are the basic building blocks of programs:

- keywords (`begin`, `end`, `while`).
- identifiers (`myVariable`, `yourType`)
- numbers (137, $6.022e23$)
- symbols ($+$, $-$)
- string literals ("Hello world")
- described (mainly) by regular grammars

Example

: identifiers

$$\begin{aligned} \text{Id} &\rightarrow \text{Letter IdRest} \\ \text{IdRest} &\rightarrow \epsilon \mid \text{Letter IdRest} \mid \text{Digit IdRest} \end{aligned}$$

Other issues: international characters, case-sensitivity, limit of identifier length

Backus-Naur Form

Backus-Naur Form (BNF) is a notation for context-free grammars:

- alternation: $\mathrm{Symb} ::= \mathrm{Letter} \mid \mathrm{Digit}$
- repetition: $\mathrm{Id} ::= \mathrm{Letter}\ \{\mathrm{Symb}\}$
  or we can use a Kleene star: $\mathrm{Id} ::= \mathrm{Letter}\ \mathrm{Symb}^*$
  for one or more repetitions: $\mathrm{Int} ::= \mathrm{Digit}^+$
- option: $\mathrm{Num} ::= \mathrm{Digit}^+[.\ \mathrm{Digit}^*]$

Note that these abbreviations do not add to the expressive power of the grammar.

Parse trees

A parse tree describes the way in which a string in the language of a grammar is derived:

- root of tree is start symbol of grammar
- leaf nodes are terminal symbols
- internal nodes are non-terminal symbols
- an internal node and its descendants correspond to some production for that non terminal
- top-down tree traversal represents the process of generating the given string from the grammar
- construction of tree from string is *parsing*

Ambiguity

If the parse tree for a string is not unique, the grammar is *ambiguous*:

$$\mathrm{E} ::= \mathrm{E}\ +\ \mathrm{E} \mid \mathrm{E}\ *\ \mathrm{E} \mid \mathrm{Id}$$

Two possible parse trees for $\mathrm{A} + \mathrm{B} * \mathrm{C}$:

- $((\mathrm{A} + \mathrm{B}) * \mathrm{C})$
- $(\mathrm{A} + (\mathrm{B} * \mathrm{C}))$

One solution: rearrange grammar:

$$\begin{array}{l} \mathrm{E} ::= \mathrm{E}\ +\ \mathrm{T} \mid \mathrm{T} \\ \mathrm{T} ::= \mathrm{T}\ *\ \mathrm{Id} \mid \mathrm{Id} \end{array}$$

Ambiguity

If the parse tree for a string is not unique, the grammar is *ambiguous*:

$$\mathrm{E} ::= \mathrm{E} \ + \ \mathrm{E} \mid \mathrm{E} \ * \ \mathrm{E} \mid \mathrm{Id}$$

Two possible parse trees for $\mathrm{A} + \mathrm{B} * \mathrm{C}$:

- $((\mathrm{A} + \mathrm{B}) * \mathrm{C})$
- $(\mathrm{A} + (\mathrm{B} * \mathrm{C}))$

One solution: rearrange grammar:

$$\begin{aligned} \mathrm{E} &::= \mathrm{E} \ + \ \mathrm{T} \mid \mathrm{T} \\ \mathrm{T} &::= \mathrm{T} \ * \ \mathrm{Id} \mid \mathrm{Id} \end{aligned}$$

Why is ambiguity bad?

Dangling else problem

Consider:

$$\text{S ::= if E then S}$$
$$\text{S ::= if E then S else S}$$

The string

$$\text{if E1 then if E2 then S1 else S2}$$

is ambiguous (Which then does else S2 match?)

Dangling else problem

Consider:

$$\text{S} ::= \text{if E then S}$$
$$\text{S} ::= \text{if E then S else S}$$

The string

$$\text{if E1 then if E2 then S1 else S2}$$

is ambiguous (Which then does else S2 match?)

Solutions:

- Pascal rule: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- grammatical solution: introduce explicit end-marker

Lexing and Parsing

Lexer

- Reads sequence of characters of input program
- Produces sequence of tokens (identifiers, keywords, numbers, ...)
- Specified by regular expressions

Parser

- Reads sequence of tokens
- Produces parse tree
- Specified by context-free grammars

Both lexers and parsers can be automatically generated from their grammars using tools such as lex/flex respectively yacc/bison.