Course Information

Online at http://cs.nyu.edu/rgrimm/teaching/sp10-pl/, including grading policy and collaboration policy.

Acknowledgments

I will draw on a number of sources throughout the semester, notably on previous versions of this course by Clark Barrett, Allan Gottlieb, Martin Hirzel, and (indirectly) Ed Osinski.

Sources for today’s lecture:

PLP, chapters 1-2.
Clark Barrett. Lecture notes, Fall 2008.
What is a Program?
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, 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).
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 plus 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?* Answer: in machine language!
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 an interpreter or compiler. An interpreter implements a (virtual) machine, whose language is the high-level language. In contrast, a compiler converts high-level programs into machine language. More on this later...

Question: How do you write an interpreter or compiler?

Answer: in assembly language (at least the first time)
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:
  \[ \text{ALGOL} \Rightarrow \text{BCPL} \Rightarrow \text{C} \Rightarrow \text{C++} \]

- **Special Purposes**: some languages are designed specifically to make a particular task easier. For example, ML 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

What makes a language successful?

- **Expressive Power**: Though most languages share the same theoretical power (i.e. they are all *Turing complete*), it is often much easier to accomplish a task in one language as opposed to another.

- **Ease of Use for Novices**: BASIC, LOGO, PASCAL, and even JAVA owe much their popularity to the fact that they are easy to learn.

- **Ease of Implementation**: Languages like BASIC, PASCAL, JAVA were designed to be easily portable from one platform to another.

- **Standardization**: C, C++, and C# all have well-defined international standards, ensuring source code portability.

- **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; 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 in particular is emphasized in the text 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, C#
  - data structures and their operations are bundled together
  - inheritance, information hiding

- **Functional (applicative):** SCHEME, LISP, ML, 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

- **Scripting:** Unix shells, PERL, PYTHON, TCL, PHP, JAVASCRIPT
  - Often used to *glue* other programs together.
Classifying Programming Languages

Hybrids

- Imperative + Object-Oriented: C++
- Functional + Logical: CURRY
- Functional + Object-Oriented: O’CAML, OHASKEE

Concurrent Programming

- Not really a category of programming languages
- Usually implemented with extensions within existing 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, ML, PROLOG,

- **Wide-spectrum** languages try to do both:
  - ADA, C++, (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
- 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 HASKELL:

```
foldr (+) 0 xs
```

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 (not C++)
- facilities for generic programming (after a while for JAVA)
- development support including editors, libraries, compilers

We will discuss many of these in detail this semester.
Compilation Overview

Major phases of a compiler:

1. **Lexer**: Text $\rightarrow$ Tokens

2. **Parser**: Tokens $\rightarrow$ Parse Tree

3. **Type checking**: Parse Tree $\rightarrow$ Parse Tree + Types

4. **Intermediate code generation**: Parse Tree $\rightarrow$ Intermed. Representation (IR)

5. **Optimization I**: IR $\rightarrow$ IR

6. **Target code generation**: IR $\rightarrow$ assembly/machine language

7. **Optimization II**: target language $\rightarrow$ target language
Syntax and Semantics

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

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

*Semantics* denotes meaning:

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

We will not be covering semantic analysis (this is covered in the compilers course), though you can read about it in chapter 4 if you are interested.

We now look at grammars in more detail.
Grammars

A grammar \( G \) is a tuple \((\Sigma, N, S, \delta)\), where:

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

\[
\text{ABC} \ldots \rightarrow \text{XYZ} \ldots
\]

where \( A, B, C, D, X, Y, Z \) 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 \( S \).
Consider the following grammar $G$:

- $N = \{S, X, Y\}$
- $S = S$
- $\Sigma = \{a, b, c\}$

$\delta$ 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 terminal and optionally a non-terminal on the right
  - Non-terminals on the right side of rules must either always precede terminals or always follow terminals
  - Recognizable by finite state automaton

- 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

- Context-sensitive grammars (Type 1)
  - Each production is of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$,
  - $A$ is a non-terminal, and $\alpha, \beta, \gamma$ are arbitrary strings ($\alpha$ and $\beta$ may be empty, but not $\gamma$)
  - Recognizable by linear bounded automaton

- Unrestricted grammars (Type 0)
  - No restrictions
  - Recognizable by turing machine
Regular Expressions

An alternate way of describing a regular language over an alphabet $\Sigma$ 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 $\Sigma$:

- $\epsilon$ denotes $\emptyset$
- a character $x$, where $x \in \Sigma$, denotes $\{x\}$
- (sequencing) a sequence of two regular expressions $RS$ denotes $\{\alpha\beta | \alpha \in [R], \beta \in [S]\}$
- (alternation) $R | S$ 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

- $R^? \equiv \epsilon | R$
- $R^+ \equiv RR^*$
Regular Grammar Example

A grammar for floating point numbers:

\[
\begin{align*}
\text{Float} & \rightarrow \text{Digits} \mid \text{Digits} . \text{Digits} \\
\text{Digits} & \rightarrow \text{Digit} \mid \text{Digit} \text{Digits} \\
\text{Digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

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

\[
\text{Id} \rightarrow \text{Letter IdRest} \\
\text{IdRest} \rightarrow \epsilon \mid \text{Letter IdRest} \mid \text{Digit IdRest}
\]

Other issues: international characters, case-sensitivity, limit of identifier length
Backus-Naur Form (BNF) is a notation for context-free grammars:

- alternation: \( \text{Symb ::= Letter} \mid \text{Digit} \)

- repetition: \( \text{Id ::= Letter } \{ \text{Symb} \} \)
  
  or we can use a Kleene star: \( \text{Id ::= Letter Symb}^* \)

  for one or more repetitions: \( \text{Int ::= Digit}^+ \)

- option: \( \text{Num ::= Digit}^+ [. \text{Digit}^*] \)

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

\[
E ::= E + E \mid E \ast E \mid \text{Id}
\]

Two possible parse trees for \(A + B \ast C\):

- \(((A + B) \ast C)\)
- \((A + (B \ast C))\)

One solution: rearrange grammar:

\[
E ::= E + T \mid T \\
T ::= T \ast \text{Id} \mid \text{Id}
\]

*Why is ambiguity bad?*
Dangling Else Problem

Consider:

\[
S ::= \text{if } E \text{ then } S \\
S ::= \text{if } E \text{ then } S \text{ else } S
\]

The string

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]

is ambiguous (Which \textbf{then} does \textbf{else } \textbf{S}_2 match?)

Solutions:

- \textbf{PASCAL} rule: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- syntactical solution: introduce explicit end-marker
Some Practical Considerations
Abstract Syntax Trees

Many non-terminals inside a parse tree are artifacts of the grammar.

Remember:

\[
E ::= E + T | T \\
T ::= T * Id | Id
\]

The parse tree for \( B \ast C \) can be written as

\[
E(T(Id(B), Id(C)))
\]

In constrast, an abstract syntax tree (AST) captures only those tree nodes that are necessary for representing the program.

In the example:

\[
T(Id(B), Id(C))
\]

Consequently, many parsers really generate abstract syntax trees.
Abstract Syntax Trees

Another explanation for *abstract syntax tree*: It’s a tree capturing only semantically relevant information for a program, i.e., omitting all formatting and comments.

Question 1: *What is a concrete syntax tree?*

Question 2: *When do I need a concrete syntax tree?*
Scannerless Parsing

Separating syntactic analysis into lexing and parsing helps performance. After all, regular expressions can be made very fast.

But it also limits language design choices. For example, it’s very hard to compose different languages with separate lexers and parsers — think embedding SQL in JAVA.

Scannerless parsing integrates lexical analysis into the parser, making this problem more tractable.

Well, to embed SQL in JAVA, we still need support for composing grammars ... (see Robert Grimm, Better Extensibility through Modular Syntax, in Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, pp. 38-51, June 2006).