Programming Languages

CSCI-GA.2110-001
Summer 2012

Dr. Cory Plock

What this course is

- A study of programming language paradigms
  - Imperitive
  - Functional
  - Logical
  - Object-oriented
- Tour of programming language history & roots.
- Introduction to core language design & implementation concepts.
- Exposure to languages/paradigms you may not have used before.
- Reasoning about language benefits/pitfalls.
- Explores programming language implementation.
- Offers an appreciation of language standards.
- Provides the ability to more quickly learn new languages.

What this course isn't

- A comprehensive study of particular languages.
- An exercise in learning as many languages as possible.
- A software engineering course.
- A compiler course.

Introduction

The main themes of programming language design and use:

- Paradigm (Model of computation)
- Expressiveness
  - control structures
  - abstraction mechanisms
  - types and their operations
  - tools for programming in the large
- Ease of use: Writeability / Readability / Maintainability

Language as a tool for thought

- Role of language as a communication vehicle among programmers can be just as important as ease of writing
- All general-purpose languages are *Turing complete* (They can compute the same things)
- But languages can make expression of certain algorithms difficult or easy.
  - Try multiplying two Roman numerals
- Idioms in language A may be useful inspiration when writing in language B.

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
```

Is this natural? *It is if you're used to it*

Programming paradigms

- *Imperative (von Neumann)*: **Fortran**, **Pascal**, **C**, **Ada**
  - programs have mutable storage (state) modified by assignments
  - the most common and familiar paradigm
- *Functional (applicative)*: **Scheme**, **Lisp**, **ML**, **Haskell**
  - functions are first-class values
  - *side effects* (e.g., assignments) discouraged
- *Logical (declarative)*: **Prolog**, **Mercury**
  - programs are sets of assertions and rules
- *Object-Oriented*: **Simula 67**, **Smalltalk**, **C++**, **Ada95**, **Java**, **C#**
  - data structures and their operations are bundled together
  - inheritance
- Functional + Logical: **Curry**
- Functional + Object-Oriented: **O'Caml**, **O'Haskell**

The Beginnings

- Before FORTRAN/COBOL/ALGOL, programs were written in *assembly*.
- FORTRAN
  - Invented by John Backus et al., released in 1957.
  - First successful high-level programming language.
  - Primary use: scientific computing and mathematics.
  - Example:

```
A = C + D
```

- COBOL
  - Designed by committee, released late 1960.
  - Common or Business-Oriented Language.
  - Data processing, business, finance, administrative systems.
  - Example:

```
ADD C TO D GIVING A
```

The Beginnings (Continued)

- ALGOL
  - Invented by a group of European & American computer scientists, released in 1958.
  - Popularized many PL concepts still in use today.
    - BNF
    - Compound statements using blocks
    - `case` statement
    - Call-by-reference
    - Orthogonality
  - Was not a commercial success (e.g., no standard I/O).

```
IF Ivar > Jvar THEN Ivar ELSE Jvar FI := 3;
```

Genealogy

- **FORTRAN** (1957) $\Rightarrow$ **Fortran90**, **HP**
- **COBOL** (1960) $\Rightarrow$ **COBOL 2000**
- **Algol60** $\Rightarrow$ **Algol68/Algol W** $\Rightarrow$ **Pascal** $\Rightarrow$ **Ada**
- **Algol60** $\Rightarrow$ **BCPL** $\Rightarrow$ **C** $\Rightarrow$ **C++**
- **Algol60** $\Rightarrow$ **Simula** $\Rightarrow$ **Smalltalk**
- **APL** $\Rightarrow$ **J**
- **Snobol** $\Rightarrow$ **Icon**
- **Lisp** $\Rightarrow$ **Scheme** $\Rightarrow$ **ML** $\Rightarrow$ **Haskell**

with lots of cross-pollination: e.g., **Java** is influenced by **C++**, **Smalltalk**, **Lisp**, **Ada**, etc.

Predictable performance vs. ease of writing

- Low-level languages mirror the physical machine:
  - **Assembly**, **C**, **Fortran**
- High-level languages model an abstract machine with useful capabilities:
  - **ML**, **Setl**, **Prolog**, **SQL**, **Haskell**
- Wide-spectrum languages try to do both:
  - **Ada**, **C++**, **Java**, **C#**
- High-level languages have garbage collection, are often interpreted, and cannot be used for real-time programming. The higher the level, the harder it is to determine cost of operations.

Common Ideas

Modern imperative 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

Language standards

Developed by working groups of standards bodies (ANSI, ISO).

- Pro: Discourages countless language flavors (ala LISP)
- Con: Places creative freedom in the hands of a few people.
- Major compiler manufacturers generally align to the standards.
- Main goal: increases portability.
- Specifies universal language syntax.
- Defines, but does **not enforce**, syntactic and semantic correctness.

Example: incorrect code, but GNU C++ compiler doesn't warn by default:

```
int x;
int y = x + 2; // x is undefined
```

Language libraries

The programming environment may be larger than the language.

- The predefined libraries are *indispensable* to the proper use of the language, *and its popularity.*
- The libraries are defined in the language itself, but they have to be internalized by a good programmer.

Examples:

- C++ standard template library
- Java Swing classes
- Ada I/O packages

Syntax and semantics

- Syntax refers to external representation:
  - Given some text, is it a well-formed program?
- Semantics denotes meaning:
  - Given a well-formed program, what does it mean?
  - Often depends on context.

  The division is somewhat arbitrary.
- Note: It *is* possible to fully describe the syntax and sematics of a programming language by syntactic means (e.g., Algol68 and W-grammars), but this is highly impractical.

  Typically use a grammar for the context-free aspects, and different method for the rest.
- Similar looking constructs in different languages often have subtly (or not-so-subtly) different meanings

Compilation overview

Major phases of a compiler:

1. lexer: text $\longrightarrow$ tokens
2. parser: tokens $\longrightarrow$ parse tree
3. semantic analyzer: parse tree $\longrightarrow$ abstract syntax tree
4. intermediate code generation
5. optimization (machine independent): local & global redundancy elimination, loop optimization
6. target code generation
7. optimization (machine dependent): instruction scheduling, register allocation, peephole optimization

Grammars

A *grammar* G is a tuple (Σ, N, S, δ)

- Σ is the set of *terminal* symbols (alphabet)
- N is the set of *non-terminal* symbols
- S is the distinguished non-terminal: the root symbol
- δ is the set of rewrite rules (productions) of the form:

  $$\mathrm{ABC} \ldots ::= \mathrm{XYZ} \ldots$$

  where $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}$ are terminals and non terminals.
- The *language* is the set of sentences containing **only** terminal symbols that can be generated by applying the rewriting rules starting from the root symbol (let's call such sentences *strings*)

BNF: notation for context-free grammars

(BNF = Backus-Naur Form) Some conventional abbreviations:

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

- abbreviations do not add to expressive power of grammar
- need convention for metasymbols – what if "|" is in the language?

Grammar Example (partial)

```
<typedecl>      ::= type <typedeflist>
<typedeflist>   ::= <typedef> [ <typedeflist> ]
<typedef>     ::= <typeid> = <typespec> ;
<typespec>    ::= <typeid> |
                  <arraydef>  |  <ptrdef>  |  <rangedef>  |
                  <enumdef>  |  <recdef>
<typeid>      ::= <ident>
<arraydef>    ::= [ packed ] array '[' <rangedef> ']' of <typeid>
<ptrdef>      ::= ^ <typeid>
<rangedef>    ::= <number> .. <number>
<number>      ::= <digit> [ <number> ]
<enumdef>     ::= ( <idlist> )
<idlist>      ::= <ident> { , <ident> }
<recdef>      ::= record <vardecllist> end ;
```

The Chomsky hierarchy

- Regular grammars (Type 3)
  - all productions can be written in the form: $\mathrm{N} ::= \mathrm{TN}$
  - one non-terminal on left side; at most one on right
- Context-free grammars (Type 2)
  - all productions can be written in the form: $\mathrm{N} ::= \mathrm{XYZ}$
  - one non-terminal on the left-hand side; mixture on right
- Context-sensitive grammars (Type 1)
  - number of symbols on the left is no greater than on the right
  - no production shrinks the size of the sentential form
- Type-0 grammars
  - no restrictions

Regular expressions

Regular expressions can be used to generate or recognize regular languages.

We say that a regular expression R denotes the language $[\![R]\!]$.

Basic regular expressions:

- 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) $R|S$ 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

Shorthands:

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

Regular grammar example

A grammar for floating point numbers:

$$\begin{aligned} \text{Float} &::= \text{Digits} \mid \text{Digits}\,.\,\text{Digits} \\ \text{Digits} &::= \text{Digit} \mid \text{Digit}\,\text{Digits} \\ \text{Digit} &::= 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

A regular expression for floating point numbers:

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

Perl offer some shorthands:

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

or

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

Lexical Issues

Lexical: formation of words or tokens.

- Described (mainly) by regular grammars
- Terminals are characters. Some choices:
  - character set: ASCII, Latin-1, ISO646, Unicode, etc.
  - is case significant?
- Is indentation significant?
  - Python, Occam, Haskell

Example: identifiers

$$\begin{aligned}\mathrm{Id} &::= \mathrm{Letter\ IdRest} \\ \mathrm{IdRest} &::= \epsilon \mid \mathrm{Letter\ IdRest} \mid \mathrm{Digit\ IdRest}\end{aligned}$$

Missing from above grammar: limit of identifier length

Parse trees

A parse tree describes the grammatical structure of a sentence

- root of tree is root 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 sentence from the grammar
- construction of tree from sentence is *parsing*

Ambiguity

If the parse tree for a sentence 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:

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

Harder problems – disambiguate these (courtesy of Ada):

- $\mathrm{function\_call} ::= \mathrm{name}\ (\mathrm{expression\_list})$
- $\mathrm{indexed\_component} ::= \mathrm{name}\ (\mathrm{index\_list})$
- $\mathrm{type\_conversion} ::= \mathrm{name}\ (\mathrm{expression})$

Dangling else problem

Consider:

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

The sentence

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

is ambiguous (Which then does $\mathrm{else\ S2}$ 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

The general ambiguity problem is unsolvable

Scanners and Parsers

- **Scanners** (or *tokenizers*) read input, identify, and extract small input fragments called tokens.
  - Identifiers
  - Constants
  - Keywords
  - Symbols: (,), [,], !, =, !=, etc.
- **Parsers** attempt to match input tokens to grammar rules.
  - **LL** (or: recursive descent, top-down) parsers are depth-first, begin at the start symbol and recurse on each RHS non-terminal.
  - **LR** (or: bottom-up) parsers

Tools for creating scanners and parsers

- **Lex** (or **Flex**) is a lexical analyzer generator.
  - Input: rules containing regular expressions.
  - Output: a lexical analyzer.
- **Yacc** (or **Bison**) is a parser generator.
  - Input: Context-free grammar and Lex input (optional).
  - Output: An LR parser.