Programming Languages

G22.2110
Summer 2008

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
Role of language as a communication vehicle among programmers is more important than 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.
■ 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
Course Goals

- **Intellectual:** help you understand benefit/pitfalls of different approaches to language design, and how they work.

- **Practical:**
  - you will probably design languages in your career (at least small ones)
  - understanding how to use a programming paradigm can improve your programming even in languages that don’t support it
  - knowing how feature is implemented helps us understand time/space complexity

- **Academic:** good start on core exam
Major phases of a compiler:

1. lexer: text $\rightarrow$ tokens
2. parser: tokens $\rightarrow$ parse tree
3. intermediate code generation
4. optimization
5. target code generation
6. optimization
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
Genealogy

- **FORTRAN (1957)** $\Rightarrow$ Fortran90, HP
- **COBOL (1956)** $\Rightarrow$ COBOL 2000
  - still a large chunk of installed software
- **Algol60** $\Rightarrow$ **Algol68** $\Rightarrow$ **Pascal** $\Rightarrow$ **Ada**
- **Algol60** $\Rightarrow$ **BCPL** $\Rightarrow$ **C** $\Rightarrow$ **C++**
- **APL** $\Rightarrow$ **J**
- **Snobol** $\Rightarrow$ **Icon**
- **Simula** $\Rightarrow$ **Smalltalk**
- **Lisp** $\Rightarrow$ **Scheme** $\Rightarrow$ **ML** $\Rightarrow$ **Haskell**

with lots of cross-pollination: e.g. **Java** is influenced by **C++**, **Smalltalk**, **Lisp**, **Ada**, etc.
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.
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 (not C++)
- facilities for generic programming (new in Java)
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
Different users have different needs:

- *programmers*: tutorials, reference manuals, programming guides (idioms)
- *implementors*: precise operational semantics
- *verifiers*: rigorous axiomatic or natural semantics
- *language designers and lawyers*: all of the above

Different levels of detail and precision

- but none should be sloppy!
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 semantics 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.
A grammar $G$ is a tuple $(\Sigma, N, S, \delta)$

- $N$ is the set of non-terminal symbols
- $S$ is the distinguished non-terminal: the root symbol
- $\Sigma$ is the set of terminal symbols (alphabet)
- $\delta$ is the set of rewrite rules (productions) of the form:

  
  \[ABC \ldots ::= XYZ \ldots\]

  where $A, B, C, D, X, Y, Z$ 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)
The Chomsky hierarchy

- **Regular grammars (Type 3)**
  - all productions can be written in the form: $N ::= TN$
  - one non-terminal on left side; at most one on right

- **Context-free grammars (Type 2)**
  - all productions can be written in the form: $N ::= XYZ$
  - 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
An alternate way of describing a regular language is with regular expressions. We say that a regular expression $R$ denotes the language $\langle R \rangle$. Recall that a language is a set of strings.

Basic regular expressions:

- $\epsilon$ denotes the empty language.
- A character $x$, where $x \in \Sigma$, denotes $\{x\}$.
- (sequencing) A sequence of two regular expressions $RS$ denotes $\{\alpha\beta \mid \alpha \in \langle R \rangle, \beta \in \langle S \rangle\}$.
- (alternation) $R|S$ denotes $\langle R \rangle \cup \langle S \rangle$.
- (Kleene star) $R^*$ denotes the set of strings which are concatenations of zero or more strings from $\langle R \rangle$.
- Parentheses are used for grouping.

Shorthands:

- $R^2 \equiv \epsilon|R$.
- $R^+ \equiv RR^*$.
A grammar for floating point numbers:

Float ::= Digits | Digits . Digits  
Digits ::= Digit | Digit Digits  
Digit ::= 0|1|2|3|4|5|6|7|8|9

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: 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

\[
\begin{align*}
  \text{Id} &::= \text{Letter} \ \text{IdRest} \\
  \text{IdRest} &::= \epsilon \ | \ \text{Letter} \ \text{IdRest} \ | \ \text{Digit} \ \text{IdRest}
\end{align*}
\]

Missing from above grammar: limit of identifier length
(BNF = Backus-Naur Form) Some conventional abbreviations:

- **Alternation:** `Symb ::= Letter | Digit`
- **Repetition:** `Id ::= Letter {Symb}`
  or we can use a Kleene star: `Id ::= Letter Symb*`
  for one or more repetitions: `Int ::= Digit+`
- **Option:** `Num ::= Digit+[. Digit*]`

- Abbreviations do not add to expressive power of grammar
- Need convention for metasymbols – what if “|” is in the language?
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
If the parse tree for a sentence is not unique, the grammar is *ambiguous*:

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

Two possible parse trees for “A + B * C”:

- ((A + B) * C)
- (A + (B * C))

One solution: rearrange grammar:

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

Harder problems – disambiguate these (courtesy of Ada):

- function_call ::= name (expression_list)
- indexed_component ::= name (index_list)
- type_conversion ::= name (expression)
Consider:

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

The sentence

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

is ambiguous (Which \text{then} does \text{else } S_2 \text{ 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