G22.2110
Programming Languages

• The main themes of programming language design and use:
  – Model of computation
  – Expressiveness
    • types and their operations
    • control structures
    • abstraction mechanisms
    • tools for programming in the large
  – Ease of use: Writeability / Readability / Maintainability
Models of Computation

• **Imperative**: programs have mutable storage (state) modified by assignments
  – by far the most common and familiar
• **Functional** (applicative): programs are pure functions
  – much use in AI, formal semantics, language research
• **Declarative**: programs are unordered sets of assertions and rules
  – Prolog, data base applications
Genealogy

- FORTRAN (1957) => Fortran90, HPF
- COBOL (1956) => COBOL2000
  - still a large chunk of installed software
- Algol60 => Algol68 => Pascal => Ada
- Algol60 => BCPL => C
- Simula => Smalltalk, C++
- LISP => Scheme, ML, Haskell
  - with plenty of cross-influences: Java inherits from C++, Smalltalk, LISP, Ada, etc.
Common Ideas

- Modern imperative languages (Ada, C++, Java) have similar characteristics:
  - large number of features (grammar with several hundred productions, 500 page reference manuals...)
  - a rich 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 (*not Java*)
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, Python
• Wide-spectrum languages try to do both, more or less well:
  – Ada, C++, Java
• High-level languages are often interpreted, have garbage collector, are dynamically typed, and cannot be used for real-time programming. Cost of operations is not directly visible.
  – Java is a hybrid
Language as a tool for thought (Iverson)

• Drawing a histogram in APL:
  
  - "* " [ V °≥ ⌊V]
  - Is it natural? (only if you happen to think that way)
  - Role of language as a communication vehicle among programmers is more important than ease of writing
  - APL is an extreme case (write-only language)
  - All languages have the same expressive power; arguments of the form “you can’t do this in X” are meaningless.
  - But.. Idioms in language A may be useful inspiration when writing in language B.
The programming environment may be larger than the language

- The predefined libraries are indispensable to the proper use of the language.
- The libraries are defined in the language itself, but they have to be internalized by a good programmer.
- The C++ standard template library
- The Java Swing classes
- The Ada I/O packages
- There is a law of conservation of junk: whether it goes into the language or in the library, the total amount of miscellaneous information is roughly the same from language to language.
Language Definition

• 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 language lawyers
    • all of the above

• Different levels of detail and precision
  – But none of them can be sloppy!
Syntax and Semantics

• Syntax refers to external representation
• Semantics denotes meaning
• Distinction is convenient but arbitrary: can describe fully a programming language by syntactic means (Algol68 and W-grammars), but resulting grammar is hard to use.
• In practice, describe context-free aspects with a grammar, the rest with prose or a different formal notation (equations, pre/post conditions, assertions).
Grammars

- A set of non-terminal symbols
- A distinguished non-terminal: the root symbol
- A set of terminal symbols
- A series of rewrite rules (productions) of the form:
  \[ \text{ABC}.. \ ::= \text{XYZ}.. \]
- where \( A, B, C, D, X, Y, Z \) 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
The Chomsky hierarchy

• Regular grammars:
  – all productions have the form: $N ::= TN$
  – (one non-terminal on each side)

• Context-free grammars:
  – all productions have the form: $N ::= XYZ$
  – (one non-terminal on the left-hand side)

• Context-sensitive languages:
  – The number of symbols on the left is no greater than on the right
  – (no production shrinks the size of the sentential form)

• Type-0 languages
  – no restrictions
Lexical Issues

- Lexical structure of programming languages is simple.
- Described mostly by regular grammar
- Terminals are characters:
  - need to specify character set: ASCII, Latin-1, ISO646, Unicode...
  - need to specify if case is significant
  - need to specify external source representation for portability
- Identifiers:
  - Id => letter
  - Id => letter Id
- Grammars can’t count: previous rule does not specify size limit
BNF: standard notation for context-free grammars

• A series of conventional abbreviations:
  – alternation:  \text{symb ::= Letter | Digit}
  – option:
    • \text{if_stat ::= IF condition THEN statement [else_part] END IF}
  – repetition:
    • \text{else_part ::= \{elseif_part\} \[ELSE statement\]}
    • \text{elsif_part ::= ELSIF condition THEN statement}
  – also noted with Kleene star:
    • \text{id ::= Letter symb*}
  – \textit{Does not add to expressive power of grammar, could be done with recursion (tastes on readability differ)}
  – \textit{need convention for metasymbols: what if ‘|’ is in the language?}

• Compare Barnes appendix 3, Stroustrup Appendix A.
Parse trees

- A parse tree describes the grammatical structure of a sentence
  - leaf nodes are terminal symbols
  - root of tree is root symbol of grammar
  - 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
  - Reconstruction of tree from sentence is parsing
Ambiguity

• If the parse tree for a sentence is not unique, the grammar is ambiguous:
  – \[ E ::= E + E \mid E \times E \mid Id \]
  – parse tree for \( A + B \times C \) ?
  – Either \((A + B) \times C\) or \((A + (B \times C))\)
  – Solution: non-terminals with different precedences
  – \[ E ::= E + T \mid T \]
  – \[ T ::= T \times Id \mid Id \]

• Harder to resolve syntactically:
  – function_call ::= name (expression_list)
  – indexed_component ::= name (index_list)
  – type_conversion ::= name (expression)
The dangling else problem

- $S ::= \text{if } E \text{ then } S$
- $S ::= \text{if } E \text{ then } S \text{ else } S$
- $\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$
  is ambiguous
- Solutions:
  - Pascal rule: $\text{else}$ matches most recent $\text{if}$
  - grammatical solution: different productions for balanced and unbalanced if-statements
  - grammatical solution: introduce explicit end-marker
- general ambiguity problem is unsolvable