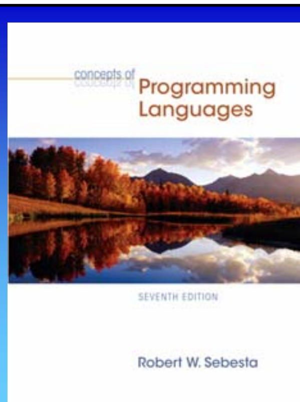
Chapter 2

Evolution of the Major Programming Languages



ISBN 0-321-33025-0

Chapter 2 Topics

1. Zuse's Plankalkul
2. Minimal Hardware Programming: Pseudocodes
3. The IBM 704 and Fortran
4. Functional Programming: LISP
5. The First Step Toward Sophistication: ALGOL 60
6. Computerizing Business Records: COBOL
7. The Beginnings of Timesharing: BASIC

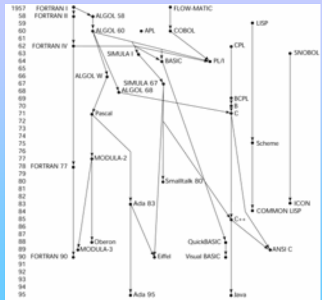
Chapter 2 Topics (continued)

8. Everything for Everybody: PL/I
9. Two Early Dynamic Languages: APL and SNOBOL
10. The Beginings of Data Abstraction: SIMULA 67
11. Orthogonal Design: ALGOL 68
12. Some Early Desscendants of the ALGOLs
13. Programming Based on Logic: Prolog
14. Hisotry's Largest Design Effort: Ada

Chapter 2 Topics (continued)

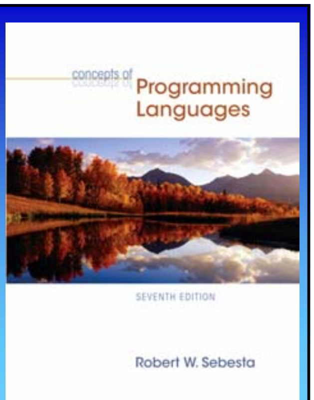
15.Object-Oriented Programming: Smalltalk
16. Combining Imperative ad Object-Oriented Features: C++
17. An Imperative-Based Object-Oriented Language: Java
18. Scripting Languages: JavaScript, PHP, and Python
19. A C-Based Language for the New Millennium: C#
20. Markup/Programming Hybrid Languages

Genealogy of Common Languages



Chapter 3

Describing Syntax and Semantics



ISBN 0-321-33025-0

Chapter 3 Topics

1. Introduction
2. The General Problem of Describing Syntax
3. Formal Methods of Describing Syntax
4. Attribute Grammars
5. Describing the Meanings of Programs: Dynamic Semantics

3.1 Introduction

- **Syntax**: the form or structure of the expressions, statements, and program units
- **Semantics**: the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

3.2 The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

- **Recognizers**
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
  - Detailed discussion in Chapter 4
- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

3.3 Formal Methods of Describing Syntax

- Backus-Naur Form and Context-Free Grammars
  - Most widely known method for describing programming language syntax
- Extended BNF
  - Improves readability and writability of BNF
- Grammars and Recognizers

Chapter 4 Topics

1. Introduction
2. Lexical Analysis
3. The Parsing Problem
4. Recursive-Descent Parsing
5. Bottom-Up Parsing

4.1 Introduction

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

4.1 Introduction (cont.)

- Reasons to use BNF to describe syntax:
  - Provides a clear and concise syntax description
  - The parser can be based directly on the BNF
  - Parsers based on BNF are easy to maintain

4.1 Introduction (cont.)

- Reasons to separate lexical and syntax analysis:
  - Simplicity – less complex approaches can be used for lexical analysis; separating them simplifies the parser
  - Efficiency – separation allows optimization of the lexical analyzer
  - Portability – parts of the lexical analyzer may not be portable, but the parser always is portable

4.2 Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a “front-end” for the parser
- Identifies substrings of the source program that belong together – lexemes
  - Lexemes match a character pattern, which is associated with a lexical category called a token
  - **`sum`** is a lexeme; its token may be **`IDENT`**

 2-16

4.2 Lexical Analysis (cont.)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram
- book only discusses approach 2

 2-17

Lexical Analysis

- Using Finite State Machines to implement lexical scan
- Example: Design a FSM which translates input text line by line so that the following rule is followed correcting spelling mistakes wrt “ei” and “cei”: *“i should be followed by e except when immediately followed by c”.*
- Input: She will eat a pie if there is a pei and when she recieves it.
- Output: She will eat a pie if there is a pie and when she receives it.

 2-18

4.2 Lexical Analysis (cont.)

- State diagram design:
  - A naïve state diagram would have a transition from every state on every character in the source language – such a diagram would be very large!

 2-19

4.2 Lexical Analysis (cont.)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a *character class* that includes all letters
  - When recognizing an integer literal, all digits are equivalent – use a *digit class*

 2-20

4.2 Lexical Analysis (cont.)

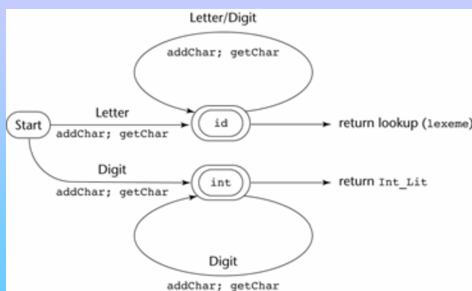
- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

 2-21

4.2 Lexical Analysis (cont.)

- Convenient utility subprograms:
  - **getChar** – gets the next character of input, puts it in **nextChar**, determines its class and puts the class in **charClass**
  - **addChar** – puts the character from **nextChar** into the place the lexeme is being accumulated, **lexeme**
  - lookup – determines whether the string in **lexeme** is a reserved word (returns a code)

State Diagram



4.2 Lexical Analysis (cont.)

Implementation (assume initialization):

```
int lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT)
      {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
       …
```

4.2 Lexical Analysis (cont.)

```
...
case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }  /* End of switch */
}  /* End of function lex */
```


HW problems for practice on FSMs

- Design a FSM that translates text in which properly delimited *the* is replaced by *a*.
- Design a FSM to strip out comments from a C or C++ program.
- Design a FSM to recognize identifiers in C.