APPENDIX B

NOTES ON LANGUAGES AND ALGORITHMS

B.1 Defining Languages with Backus–Naur Form (BNF)

In this book we define several languages, including the languages of propositional logic (235), first-order logic (276), and a subset of English (page 893). A formal language is defined as a set of strings where each string is a sequence of symbols. The languages we are interested in consist of an infinite set of strings, so we need a concise way to characterize the set. We do that with a **grammar**. The particular type of grammar we use is called a **context-free grammar**, because each expression has the same form in any context. We write our grammars in a formalism called **Backus–Naur form (BNF)**. There are four components to a BNF grammar:

Context-free grammar
Backus–Naur form (BNF)

- A set of **terminal symbols**. These are the symbols or words that make up the strings of the language. They could be letters (**A, B, C,** ...) or words (**a, aardvark, abacus,** ...), or whatever symbols are appropriate for the domain.
  Terminal symbol
- A set of **nonterminal symbols** that categorize subphrases of the language. For example, the nonterminal symbol *NounPhrase* in English denotes an infinite set of strings including "you" and "the big slobbery dog."
  Nonterminal symbol
- A **start symbol**, which is the nonterminal symbol that denotes the complete set of strings of the language. In English, this is *Sentence*; for arithmetic, it might be *Expr*, and for programming languages it is *Program*.
  Start symbol
- A set of **rewrite rules**, of the form $LHS \rightarrow RHS$, where *LHS* is a nonterminal symbol and *RHS* is a sequence of zero or more symbols. These can be either terminal or nonterminal symbols, or the symbol ϵ, which is used to denote the empty string.
  Rewrite rules

A rewrite rule of the form

$$Sentence \rightarrow NounPhrase\ VerbPhrase$$

means that whenever we have two strings categorized as a *NounPhrase* and a *VerbPhrase*, we can append them together and categorize the result as a *Sentence*. As an abbreviation, the two rules ($S \rightarrow A$) and ($S \rightarrow B$) can be written ($S \rightarrow A \mid B$). To illustrate these concepts, here is a BNF grammar for simple arithmetic expressions:

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ Operator\ Expr \mid (\ Expr\ ) \mid Number \\ Number & \rightarrow & Digit \mid Number\ Digit \\ Digit & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \\ Operator & \rightarrow & + \mid - \mid \div \mid \times \end{array}$$

We cover languages and grammars in more detail in Chapter 24. Be aware that other books use slightly different notations for BNF; for example, you might see ⟨*Digit*⟩ instead of *Digit* for a nonterminal, 'word' instead of **word** for a terminal, or ::= instead of → in a rule.

B.2 Describing Algorithms with Pseudocode

Pseudocode

The algorithms in this book are described in **pseudocode**. Most of the pseudocode should be familiar to programmers who use languages like Java, C++, or especially Python. In some places we use mathematical formulas or ordinary English to describe parts that would otherwise be more cumbersome. A few idiosyncrasies should be noted:

- **Persistent variables**: We use the keyword **persistent** to say that a variable is given an initial value the first time a function is called and retains that value (or the value given to it by a subsequent assignment statement) on all subsequent calls to the function. Thus, persistent variables are like global variables in that they outlive a single call to their function; but they are accessible only within the function. The agent programs in the book use persistent variables for *memory*. Programs with persistent variables can be implemented as *objects* in object-oriented languages such as C++, Java, Python, and Smalltalk. In functional languages, they can be implemented by *functional closures* over an environment containing the required variables.
- **Functions as values**: Functions have capitalized names, and variables have lowercase italic names. So most of the time, a function call looks like FN(x). However, we allow the value of a variable to be a function; for example, if the value of the variable f is the square root function, then $f(9)$ returns 3.
- **Indentation is significant**: Indentation is used to mark the scope of a loop or conditional, as in the languages Python and CoffeeScript, and unlike Java, C++, and Go (which use braces) or Lua and Ruby (which use **end**).
- **Destructuring assignment**: The notation "$x, y \leftarrow pair$" means that the right-hand side must evaluate to a two-element collection, and the first element is assigned to x and the second to y. The same idea is used in "**for** x, y **in** *pairs* **do**" and can be used to swap two variables: "$x, y \leftarrow y, x$"
- **Default values for parameters**: The notation "**function** F(x,y=0) **returns** a number" means that y is an optional argument with default value 0; that is, the calls F(3, 0) and F(3) are equivalent.

Generator

- **yield**: a function that contains the keyword **yield** is a **generator** that generates a sequence of values, one each time the **yield** expression is encountered. After yielding, the function continues execution with the next statement. The languages Python, Ruby, C#, and Javascript (ECMAScript) have this same feature.
- **Loops**: There are four kinds of loops:
  - "**for** x **in** c **do**" executes the loop with the variable x bound to successive elements of the collection c.
  - "**for** i = 1 to n **do**" executes the loop with i bound to successive integers from 1 to n inclusive.
  - "**while** *condition* **do**" means the condition is evaluated before each iteration of the loop, and the loop exits if the condition is false.

  - "**repeat** ... **until** *condition*" means that the loop is executed unconditionally the first time, then the condition is evaluated, and the loop exits if the condition is true; otherwise the loop keeps executing (and testing at the end).

- **Lists**: $[x, y, z]$ denotes a list of three elements. The "+" operator concatenates lists: $[1,2] + [3,4] = [1,2,3,4]$. A list can be used as a stack: POP removes and returns the last element of a list, TOP returns the last element.
- **Sets**: $\{x, y, z\}$ denotes a set of three elements. $\{x : p(x)\}$ denotes the set of all elements x for which $p(x)$ is true.
- **Arrays start at 1**: the first index of an array is 1 as in usual mathematical notation (and in R and Julia), not 0 (as in Python and Java and C).

B.3 Online Supplemental Material

The book has a Web site with supplemental material, instructions for sending suggestions, and opportunities for joining discussion lists:

- `aima.cs.berkeley.edu`

The algorithms in the book, and multiple additional programming exercises, have been implemented in Python and Java (and some in other languages) at the online code repository, accessible from the Web site and currently hosted at:

- `github.com/aimacode`