Samuel N. Kamin

Programming Languages
An Interpreter-Based Approach

Programming Languages
An Interpreter-Based Approach

Samuel N. Kamin
University of Illinois at Urbana-Champaign

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

The APLitalic font was designed by Joey Tuttle and copyrighted by I.P. Sharp Associates.

Library of Congress Cataloging-in-Publication Data

Kamin, Samuel N.
Programming Languges.

Bibliography: P.
Includes index.
1. Programming languages (electronic computers). 2. Interpreters (computer programs).
I. Title.
QA76.7.K35 1990 005.13'3 88-34998
ISBN 0-201-06824-9

Reprinted with corrections May, 1990.

BCDEFGHIJ-MA-943210

For Judy and Rebecca

Preface

The chief goal of my work as educator and author is to help people learn to write *beautiful programs.*

Donald E. Knuth, *Computer Programming as an Art*, 1974 ACM Turing Award Lecture

Beauty in programs, as in poetry, is not language-independent. There is an aesthetic dimension in programming that is visible only to the multi-lingual programmer. My purpose in writing this book has been to present that dimension in the plainest possible way.

My focus in this endeavor has been on non-imperative languages, since they present a stark aesthetic contrast with what the reader is assumed already to know — namely, PASCAL. A study of the comparatively subtle differences among more traditional languages would be less likely to make a strong impression. My specific choice, reflecting the principal concepts underlying non-imperative languages generally, is: LISP, APL, SCHEME, SASL, CLU, SMALLTALK, and PROLOG.

The beauty in these languages reveals itself slowly. Not only are the conceptual difficulties great, but they are compounded by the usual problems of learning new syntax, memorizing function and procedure names, dealing with a compiler, and other such minutiae. The challenge is to present clearly what is aesthetically interesting, and to avoid what is not.

I have tried to meet this challenge directly, in this way: Each language is presented in a syntactically and semantically simplified form, and for each we provide an interpreter, written in PASCAL. The more educated reader will be struck by this immediately, as the sample programs appear quite different from those in the real languages. No doubt some will think too much has been omitted, but it is my hope and intention that the central concepts in each language have been preserved.

The use of interpreters for simplified language subsets contributes to the overall pedagogical style, which is characterized by these principles:

Concreteness. I have emphasized the presentation of specific programs in each language. (If I were attempting to instill an appreciation of poetry, I would do it not with abstract principles or history, but with poems, and the same principle applies here, albeit on a more prosaic level.) Moreover, the languages can be described very specifically, and can be *completely understood.*

Multiple presentations of concepts. Each language is described both "top-down," by syntactic and semantic descriptions and sample programs, and "bottom-up" by the language's interpreter. Each chapter has a section documenting its interpreter, which is then listed in an Appendix.

Exploiting prior knowledge. The use of interpreters is a way of building on the reader's knowledge of PASCAL to teach him new languages. PASCAL is also used to illustrate concepts such as type-checking and scope.

Generalities emerge from specifics. Readers will learn about recursion by seeing many examples of LISP functions, about higher-order functions by seeing them used in SCHEME, and so on. (The exercises aim at making the concepts still more concrete.) Furthermore, the use of interpreters tends to wash out many nonessential distinctions among the languages covered, allowing unifying principles to reveal themselves.

Overview

The ACM Curriculum 78 recommendations (Austing, et.al. [1979]) describe the course CS8, *Organization of Programming Languages*, as "an applied course in programming language concepts." The current book is intended as a text for CS8, as well as for self-study. It takes a "comparative" approach, which is to say that it covers specific languages, teaching general concepts only as they arise in particular languages. Each of the first eight chapters covers a different language; this coverage includes a complete context-free syntax (easy because the languages are highly simplified), an interpreter written in PASCAL, and, of course, many sample programs. Chapters 9 and 10 cover, respectively, compilation and memory management.

Chapter 1 discusses the design of a simple interpreter for a language with LISP-like syntax (though without LISP data structures), with the complete PASCAL code given in Appendix B. The principal design criterion of this and all the interpreters is simplicity. The student is assumed to have a good knowledge of PASCAL. The only warning to be given is that our code makes heavy use of recursive procedures; the student whose knowledge of recursion is weak will need to spend extra time studying the Appendix. On the other hand, it is not expected that students have had much experience *writing* recursive procedures; the study of LISP is largely intended to provide such experience.

The languages covered in Chapters 2 through 8 are, in order, LISP, APL, SCHEME, SASL, CLU, SMALLTALK, and PROLOG. Each of these chapters has roughly the same structure:

- General introduction to the language, its history, and its influence.
- Discussion of the basic ideas of the language, its syntax and semantics, and simple examples.

- The design of the interpreter for the language; some code is listed here, some in the Appendix.
- One or more larger (several page) examples.
- Aspects of the language as it is in real life, including its true syntax and interesting language features not included in our version. This section is not intended for use as a reference, nor to give a complete definition of the language, but merely to provide the student with a "reading knowledge" of the language.
- A brief summary, suggestions for further study, and a glossary of terms commonly associated with the language.
- Exercises, divided in two parts: programs in the language, and modifications to the interpreter.

Some chapters have additional sections exploring concepts pertinent to that language or describing different languages based on similar concepts.

Listings of all the interpreters, in PASCAL, are available in machine-readable form from the author (see below). The individual instructor may wish to use them in different ways. Some possibilities are:

- Have students study interpreters for, and write programs in, all or a subset of the languages covered. Studying the interpreters will most likely take the form of assigning some of the interpreter-modification exercises.
- Have each student study an interpreter for only one or two of the languages, and write programs in some or all of the remaining languages.
- Use the interpreters as "black boxes;" that is, do not study the interpreters *per se*. In that case, this book has the advantage of maintaining some syntactic uniformity across languages, thereby focusing more attention on the essential features of each language, less on their syntax or other idiosyncracies.

The "prerequisite structure" of the chapters is summarized in this chart:

$$\begin{tikzcd}
 & \text{APL} & \text{CLU} \arrow[r] & \text{SMALLTALK} & \\
\text{Chapter 1} \arrow[ur] \arrow[urr] \arrow[rr] \arrow[drr] \arrow[ddrr] & & \text{LISP} \arrow[u, dashed] \arrow[d, dashed] \arrow[r] \arrow[dr] & \text{SCHEME} \arrow[r] \arrow[ddr] & \text{SASL} \\
 & & \text{PROLOG} & \text{Chapter 10} & \\
 & & \text{Chapter 9, section 9.3–9.6} \arrow[rr] & & \text{Chapter 9 part 2}
\end{tikzcd}$$

LISP is a "weak prerequisite" for CLU and PROLOG, in that some examples in those chapters use lists and recursion in LISP-like ways.

Obtaining the Interpreters

The interpreter code, and all the code appearing in this book, can be obtained by "anonymous ftp": `ftp` to node `uihub.cs.uiuc.edu`, sign on as `anonymous`, giving your own name as the password, then `cd` to directory `uiuc/kamin.distr`. You may either copy the compressed tar file `distr.tar.Z` (then uncompress and de-tar to get the `distr` directory), or simply copy the `distr` directory. For questions or comments, contact the author at: Computer Science Dept., Univ. of Illinois, 1304 W. Springfield, Urbana, IL 61801, or by electronic mail at: `kamin@cs.uiuc.edu`.

Acknowledgments

I have benefited from the comments and criticisms of many colleagues and students at the University of Illinois. The comments of Luddy Harrison, Tim Kraus, and Vipin Swarup have affected virtually every page. Ralph Johnson gave the manuscript a thorough reading. Others here whose help has been highly appreciated are (in alphabetical order) Subutai Ahmad, Nachum Dershowitz, Ken Forbus, Alan Frisch, Simon Kaplan, Kyung Min, Uday Reddy, Ed Reingold, Hal Render, Vince Russo, and Dong Tang.

Among the outside reviewers, I would especially like to thank Andrew Appel and Ryan Stansifer for their useful feedback and their continuing interest in the project. Other helpful reviews were provided by Neta Amit, Myla Archer, Ray Ford, Takayuki Dan Kimura, William J. Pervin, and Clifford Walinsky. Thanks also to Dave Jackson, Jim DeWolf, and Helen Wythe of Addison-Wesley and Lori Pickert of Archetype Publishing.

Writing a book is a strange mixture of pain and pleasure. For easing the pain and enhancing the pleasure, I thank my wife Judy.

S.N.K.

Contents

Part I
Starting Off

The idea of this book is to learn about programming languages both by programming in them and by studying interpreters for them.

In Part I, then, we present the language and interpreter which will be used as the basis for the languages and interpreters in the remainder of the book. The language we interpret here is intended to be about the minimal language, using the simplest syntax, which contains enough features to be called a programming language. The programming environment is, likewise, skeletal, providing only the ability to enter programs (no editing) and run them. Finally, the interpreter is built for simplicity, and to this goal all concerns of efficiency have been sacrificed.

My fondest hope is that the student will feel moved to correct these various deficiencies, in both this and subsequent chapters. Suggestions for improvement are made throughout the book, especially in the programming exercises that end each chapter.

Chapter 1

The Basic Evaluator

This chapter describes a simple language whose constructs should be regarded as simplified versions of the constructs of PASCAL, written in a syntax designed for ease of parsing. We then present the interpreter for that language, giving the code in PASCAL in Appendix B. The chapter ends with an outline of what this book is about and why.

1.1 The Language

Our interpreter is interactive. The user will enter two kinds of inputs: function definitions, such as:

```
(define double (x) (+ x x))
```

and expressions, such as:

```
(double 5).
```

Function definitions are simply "remembered" by the interpreter, and expressions are evaluated. "Evaluating an expression" in this language corresponds to "running a program" in most other languages.

The subsections of this section present the language's syntax, its semantics, and examples. It is to be hoped that the reader will soon find the syntax, if not elegant, at least not a major hindrance.

1.1.1 SYNTAX

The syntax of our language is like that of LISP, and can be very simply defined:[1]

```
input    ⟶  expression | fundef
fundef   ⟶  ( define function arglist expression )
```

[1]Appendix A gives an explanation of this notation.

```
arglist    ⟶  ( variable* )
expression ⟶  value
           |  variable
           |  ( if expression expression expression )
           |  ( while expression expression )
           |  ( set variable expression )
           |  ( begin expression+ )
           |  ( optr expression* )
optr       ⟶  function | value-op
value      ⟶  integer
value-op   ⟶  + | - | * | / | = | < | > | print
function   ⟶  name
variable   ⟶  name
integer    ⟶  sequence of digits, possibly preceded by minus sign
name       ⟶  any sequence of characters not an integer, and not
              containing a blank or any of the following charac-
              ters: ( ) ;.
```

A function cannot be one of the "keywords" `define`, `if`, `while`, `begin`, or `set`, or any of the value-op's. Aside from this, names can use any characters on the keyboard. Comments are introduced by the character '`;`' and continue to the end of the line; this is why '`;`' cannot occur within a name. A session is terminated by entering "`quit`"; thus, it is highly inadvisable to use this name for a variable.

Expressions are fully parenthesized. Our purpose is to simplify the syntax by eliminating such syntactic recognition problems as operator precedence. Thus, the PASCAL assignment

$$i := 2{*}j + i - k/3;$$

becomes

```
(set i (- (+ (* 2 j) i) (/ k 3))).
```

The advantage is that the latter is quite trivial to parse, where the former is not. Our form may be unattractive, but then most PASCAL programmers will agree that assignments of even this much complexity occur rarely.

1.1.2 SEMANTICS

The meanings of expression's are presented informally here (and more formally in Section 1.2.8). Note first that integers are the only values; when used in conditionals (`if` or `while`), zero represents false and one (or any other nonzero value) represents true.

(`if` e_1 e_2 e_3) — If e_1 evaluates to zero, then evaluate e_3; otherwise evaluate e_2.

`(while` e_1 e_2`)` — Evaluate e_1; if it evaluates to zero, return zero; otherwise, evaluate e_2 and then re-evaluate e_1; continue this until e_1 evaluates to zero. (A `while` expression always returns the same value, but that really doesn't matter since it is being evaluated only for its side-effects.)

`(set x e)` — Evaluate `e`, assign its value to variable `x`, and return its value.

`(begin` $e_1 \ldots e_n$`)` — Evaluate each of $e_1 \ldots e_n$, in that order, and return the value of e_n.

`(f` $e_1 \ldots e_n$`)` — Evaluate each of $e_1 \ldots e_n$, and apply function `f` to those values. `f` may be a value-op or a user-defined function; if the latter, its definition is found and the expression defining its body is evaluated with the variables of its arglist associated with the values of $e_1 \ldots e_n$.

`if`, `while`, `set`, and `begin` are called *control operations.*

All the value-op's take two arguments, except `print`, which takes one. The arithmetic operators `+`, `-`, `*`, and `/` do the obvious. The comparison operators do the indicated comparison and return either zero (for false) or one (for true). `print` evaluates its argument, prints its value, and returns its value.

As in PASCAL, there are global variables and formal parameters. When a variable reference occurs in an expression at the top level (as opposed to a function definition), it is necessarily global. If it occurs within a function definition, then if the function has a formal parameter of that name, the variable is that parameter, otherwise it is a global variable. This corresponds to the *static scope* of PASCAL, in the (relatively uninteresting) case where there are no nested procedure or function declarations.[2] There are no local variables *per se*, only formal parameters.

1.1.3 EXAMPLES

As indicated earlier, the user enters function definitions and expressions interactively. Function definitions are stored; expressions are evaluated and their values are printed. Our first examples involve no function definitions. "->" is the interpreter's prompt; an expression following a prompt is the user's input, and everything else is the interpreter's response:

```
-> 3
3

-> (+ 4 7)
11

-> (set x 4)
4
```

[2]For more on static scope, see Section 4.7 (the chapter on SCHEME), and also Chapter 9.