# PARTIAL COMPUTATION AND THE CONSTRUCTION OF LANGUAGE PROCESSORS

Frank G. Pagan

PRENTICE HALL SOFTWARE SERIES

# PARTIAL COMPUTATION AND THE CONSTRUCTION OF LANGUAGE PROCESSORS

FRANK G. PAGAN
*California State University, Fullerton*

PRENTICE HALL SOFTWARE SERIES

Brian Kernighan, Series Editor

PRENTICE HALL, Englewood Cliffs, NJ 07632

**Library of Congress Cataloging-in-Publication Data**
Pagan, Frank G.
Partial computation and the construction of language processors / Frank G. Pagan.
p. cm. — (Prentice Hall software series)
Includes bibliographic references
ISBN 0-13-651415-4
1. Electronic digital computers—Programming. 2. Programming languages (Electronic computers) I. Title. II. Series.
QA76.6.P329 1991
005.1—dc20 90-33735
CIP

Editorial supervision and interior design: Ocean View Technical Publications
Manufacturing buyer: Lori Bulwin

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-651415-4

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

# PREFACE

My main motivation for writing this book has been my conviction that the concept of partial computation is greatly underappreciated and ought to occupy a much higher level of awareness among computer scientists and computing professionals in general. Although partial computation is intimately connected with such basic concepts as binding time, efficiency, time/space trade-offs, and compilation versus interpretation, relatively few people have studied it. Most writings on the subject have been theory-oriented research papers.

To combat this, I have chosen to emphasize a practical, down-to-earth programming methodology that is based on the concept of partial computation but does not depend upon the availability of a "partial evaluator" software tool. This methodology is readily available to all programmers. As a systematic approach to the construction of program generators, it can be used to realize many of the benefits of full-blown partial computation.

The methodology is a general one and is applicable to many areas of programming. The construction of processors for programming languages is an area that is especially rich in opportunities to apply the methodology. For that reason, much of the book has to do with various aspects of language processing. It should be understood, however, that the primary subject matter is the partial-computation-based programming methodology; language processors are being used only as a context for case studies. If this book were viewed as a text on language processing, it would have to be regarded as quite incomplete and unconventional in its approach.

I do believe, nonetheless, that some of the concepts presented here should be taught to all students of language processor techniques: In that respect the book could serve as a supplement to the main text in a college-level course on compiler design. Of particular interest in this context is the interpreter-to-compiler conversion technique developed in Chapters 8 through 10. In a term project, students can use this technique to construct a compiler without being familiar with machine language.

Pascal (specifically, Borland International's Turbo Pascal 4.0) is used as the implementation language throughout the book, not because the methodology is in any way limited to that language, but simply because Pascal seems to be the closest thing to a *lingua franca* of programming languages at the present time. The book assumes that the reader, in addition to an understanding of Pascal, has a good knowledge of data structures, such as stacks and trees, and recursive programming. A prior knowledge of language processor techniques is not required.

Many of the chapters end with one or more suggestions for projects, which are often analogous to the case studies in the chapters themselves. Some of these projects would take days or weeks to complete and could be assigned by instructors as major lab problems.

It is a pleasure to acknowledge the contribution of John Jure to this work. As part of his Master's project, he labored long and hard to generate the LR(1) parse table for Cal used in Chapter 6.

*Frank G. Pagan*

# CONTENTS

# LISTINGS

1

# PROGRAMMING LANGUAGES AND THEIR SPECIFICATION

This book is about a practical programming methodology based on a largely theoretical concept known as partial computation. To undertake a study of an advanced programming methodology, one must already be proficient at programming in one or more programming languages; throughout the book, a version of Pascal will be used. As a convenient framework for case studies of application of the methodology, some aspects of the construction of processors for programming languages will be considered. Accordingly, we begin by reviewing some relevant technical properties of programming languages and their description.

## 1.1 ASPECTS OF PROGRAMS AND LANGUAGES

In this book, the most general sense of the term "language" will cover all formal notations for the expression of computer programs. One of the most basic ways of categorizing these notations is by *level*:

machine languages
assembly languages
high-level languages
fourth-generation (very high-level) languages

Much of the book, and the rest of this chapter, is concerned with the high-level languages, of which Pascal is one.

In a complete description of a programming language, there are at least four aspects that must be addressed:

lexical properties
syntax
static semantics
dynamic semantics

The *lexical* aspect has to do with the grouping of a program's individual characters into basic symbols known as *tokens*. The Pascal statement

```
while tab [i ] <= val do
  i := i+15
```

consists of a sequence of 13 tokens:

| | | |
|---|---|---|
| 1. `while` | 2. identifier `tab` | 3. `[` |
| 4. identifier `i` | 5. `]` | 6. `<=` |
| 7. identifier `val` | 8. `do` | 9. identifier `i` |
| 10. `:=` | 11. identifier `i` | 12. `+` |
| 13. numeral `15` | | |

Each token consists of one or more consecutive characters on the same line. Consecutive tokens may be separated by any number of blanks, line breaks, and comments. For most pairs of tokens, these separators are optional, but in a few places, such as between **while** and **tab** and between **val** and **do**, at least one separator must be present.

Since the number of different possibilities for identifier tokens is virtually unlimited, it is standard practice to lump together all identifiers as a single kind of token. A few other token families, such as numerals and literal strings, are treated in a similar manner. We will refer to these token types as *variable-length* tokens. The great majority of a language's tokens, including reserved words such as **while** and **do**, are *fixed-length* tokens.

When we ask about the legality of a given sequence of tokens, we move into the realm of *syntax*. The syntax of a programming language is often specified with the aid of a *context-free* or *BNF grammar*. There are many notational variations for BNF grammars, and there is no reason to try to review all of them here. We will simply employ a notation that is reasonably common and that is convenient for our purposes.

Three main kinds of symbol are used in a BNF grammar: terminal symbols, nonterminal symbols, and metasymbols. The *terminal symbols* correspond to the tokens of the programming language. Fixed-length tokens are represented by themselves, enclosed in double quotes. Variable-length tokens are represented by special words enclosed in angle brackets:

| | |
|---|---|
| **<idr>** | identifier |
| **<int>** | integer numeral |
| **<real>** | real numeral |
| **<str>** | literal string |

*Nonterminal symbols* are names of syntactic categories, such as **statement** and **term**. Each nonterminal must be defined by a *production rule* consisting, in the simplest case, of a *left part* and a *right part* separated by the metasymbol =. The left part consists of the nonterminal being defined, and the right part consists of a sequence of terminals and/or nonterminals, as in the following example:

```
whilest = "while" expr "do" stmt
```

Production rules with the same left part can be combined by using the metasymbol | to separate the right parts:

```
ifst = "if" expr "then" stmt |
       "if" expr "then" stmt "else" stmt
```

The metasymbols = and | can be read as "consists of" and "or," respectively.

For a given left part, one of the right parts can consist of 0 symbols. We will use the dummy symbol **<empty>** to indicate this phenomenon. The following rules would be an alternative way of defining the syntax of **if** statements:

```
ifst = "if" expr "then" stmt elsopt
elsopt = "else" stmt | <empty>
```

A complete grammar consists of the production rules for all the nonterminals involved in it. The nonterminal for the most inclusive syntactic category, such as **program**, is said to be the grammar's *start symbol*. The production rules will, in all likelihood, constitute a mutually recursive set of definitions; this is evident as soon as we add a rule of the form

```
stmt = whilest | ifst | ...
```

to those given previously.

Sometimes a rule with two or more right parts is directly recursive. For example,

```
stmtlist = stmt | stmtlist ";" stmt
```

has the effect of specifying that a statement list consists of one or more statements, separated by semicolons if there is more than one. It is said to be a *left recursive* rule, since the nonterminal being defined appears as the leftmost symbol in one of the right parts. The equivalent rule

```
stmtlist = stmt | stmt ";" stmtlist
```

is a *right recursive* rule.

Given a string of grammar symbols (i.e., terminal and nonterminal symbols), the process of replacing a nonterminal with one of its right parts is called a *derivation step*. A sequence of derivation steps, such as the following, is called a *derivation.*

```
   stmt
=> whilest
=> "while" expr "do" stmt
=> "while" expr "do" ifst
=> "while" expr "do" "if" expr "then" stmt
=> ...
```

The last string in a derivation is said to be *derivable* from the first string. If a final string consists entirely of terminals, it is said to be a *terminal string*. In the absence of an indication to the contrary, the initial string of a derivation will be assumed to consist only of the start symbol. A terminal string derivable from the start symbol **program** would be a particular token sequence constituting a program. If no derivation exists for a given token sequence, that token sequence is syntactically invalid. Given a grammar and a token sequence, the process of attempting to reconstruct a derivation of the latter in accordance with the former is known as *parsing* or *syntactic analysis*.

A derivation is a *leftmost derivation* (respectively, *rightmost derivation*) if at each step the nonterminal farthest to the left (respectively, right) is the one that is replaced. It can be proved that, if a token sequence is syntactically valid, it has both a leftmost derivation and a rightmost derivation. It can also be proved that any token sequence with more than one leftmost (respectively, rightmost) derivation also has more than one rightmost (respectively, leftmost) derivation. (There would then also be more than one "parse tree," but we will not be using parse trees in this book.) In that case, both the token sequence and the grammar are said to be *ambiguous*.

The grammar notations mentioned up to this point amount to what is sometimes called the "pure" version of BNF. Many parsing methods rely on the existence of pure BNF grammars that are unambiguous and that obey various other restrictions. "Extended" versions of BNF, on the other hand, make use of additional metasymbols to improve the clarity and conciseness of grammars.

The most common of these additional metasymbols are brackets and braces, used pairwise within individual right parts. Brackets indicate that the enclosed string of grammar symbols is optional. Thus we have a third way of defining the syntax of **if** statements:

```
ifst = "if" expr "then" stmt ["else" stmt]
```

Braces mean that the enclosed string is to be repeated zero or more times. They provide an alternative to recursion in many instances, such as the specification of statement lists:

```
stmtlist = stmt {";" stmt}
```

BNF grammars cannot express all aspects of the legality of token sequences. The sequence

```
while tab[i] <= val ...
```

for example, is erroneous if **tab** is not an array or is a multi-dimensional array, or if **i** is not of the array's index type, or if **val** was not declared, and so on. Although these properties of programs can be placed under the heading of *context-sensitive syntax*, implementors of language processors commonly use the term *static semantics* when speaking of them. The portions of a language description concerned with static semantics take the form of restrictions on the token sequences allowed by the grammar. The checking for violations of these restrictions can be carried out in conjunction with the syntactic analysis.

Given that a program is legal with respect to syntax and static semantics, its meaning is determined by the programming language's *dynamic semantics*. Programming-language theorists have devised various ways of formulating and formalizing dynamic semantics, but the only kind we will consider here is *operational semantics*, specified informally. Imagining that there exists a computer ideally suited to the direct execution of programs expressed in the language, we describe what that computer would do in order to execute or evaluate each kind of language construct. We might say, for example, that the construct `while E do S`, where `E` is any Boolean expression and `S` is any statement, is executed by performing the following actions:

1. `E` is evaluated, producing a result of True or False. If the result is False, nothing further is done. Otherwise,
2. `S` is executed and all is repeated from step (1).

How `E` is evaluated and how `S` is executed would be specified in other parts of the language description.

## 1.2 THE MINILANGUAGE CAL

To provide a miniature programming language that will serve as an example in later chapters, we now introduce a language named Cal. In the following example of a Cal program, the line numbers at the left have been added for reference purposes only:

```
 1 int num; int sumdiv; int d; int half; char cr.
 2 inint --> num;
 3 loop num + 1 --> num; num / 2 --> half;
 4     1 --> sumdiv; 1 --> d;
 5     loop d + 1 --> d
 6         *** while d <= half ***
 7            sumdiv + (/num/d*d = num // d // 0/) --> sumdiv
 8     end
 9   *** while sumdiv /= num ***
10 end;
11 #13 --> cr;
12 display num, cr, #(@cr - 3).
```

The only data types in Cal are integer and character. In line 1, **num**, **sumdiv**, **d**, and **half** are declared as integer variables, and **cr** is declared as a character variable. Assignment statements are marked by the token **-->** and are written with the destination variable at the end, not at the beginning as in Pascal. Execution of the statement in line 2 will change the value of the variable **num**. The expression for the new value consists of the reserved word **inint**, which has the following peculiar property: When evaluated, it waits for an integer to be read from an input device and then yields that integer. Input of character data works in a similar way, using the word **inch**.

Lines 3 through 10 constitute a loop, with the termination test at the end of its body in line 9. The termination comparison is delimited by the token pair ***** while** on the left and the token ******* on the right. Within this loop, after the four assignment statements in lines 3 and 4, there is an inner loop extending from line 5 to line 8. Its termination test is in the middle of its body, in line 6.

The assignment statement in line 7 contains a *conditional expression* of the form `(/ C // E1 // E2 /)`, where `C` is a comparison and `E1` and `E2` are expressions. The value of such a conditional expression is the value of `E1` if `C` is true and the value of `E2` if `C` is false.

Cal provides the usual four arithmetic operators, with multiplication and division taking precedence over addition and subtraction. The available comparison operators are =, <, <=, and /=. The unary operator # takes an integer and yields the corresponding ASCII character; thus line 11 assigns the carriage-return character (ASCII code 13) to **cr**. The unary operator @ takes a character and yields the corresponding integer. The **display** statement in line 12 outputs the integer value of **num**, a carriage return, and a line feed (code 10).

What does this Cal program do? A "perfect number" is an integer that is equal to the sum of all its exact divisors, including 1 but excluding itself. Since 6 = 1 + 2 + 3, 6 is a perfect number. Given an integer as input, the program computes and outputs the smallest perfect number greater than that integer. For example, given 6 as input, the program will produce 28 as the output.

The remainder of this section consists of a semiformal specification of the Cal language organized around the production rules of an extended BNF grammar. Under each rule, any associated points of semantics are stated in precise and concise English. SS stands for static semantics, and DS stands for dynamic semantics. A similar specification of another language, FCProcs, is given in the appendix.

```
program = decl {";" decl} "." series "."
```

DS: The **series** is executed.

```
decl = type <idr>
```

SS: The **<idr>** cannot be the same as a reserved word or another declared identifier.

```
type = "int" | "char"
series = stmt {";" stmt}
```

DS: The constituent **stmt**s are executed in left-to-right order.

```
stmt = asmtst | loopst | outst
asmtst = expr "-->" <idr>
```

SS: The **<idr>** must have been declared as a variable of the same type as the **expr**.

DS: The **expr** is evaluated and the result placed in the memory location allocated to the **<idr>**.

```
outst = "display" expr {"," expr}
```

DS: Each **expr** is evaluated in turn, and its result sent to an output device.

```
loopst = "loop" [series] "***" "while" comp "***" [series]
         "end"
```

DS: The first **series**, if present, is executed. The **comp** is evaluated. If it is true, the second **series**, if present, is executed and the entire process is repeated.

```
comp = expr relopr expr
relopr = "=" | "<" | "<=" | "/="
```

SS: Both **expr**s must be of type integer.

DS: **E1 R E2**—**E1** is evaluated to an integer N1. **E2** is evaluated to an integer N2. If N1 relates to N2 according to **R**, the **comp** is true; otherwise, it is false. = means “is equal to”; < means “is less than”; <= means “is less than or equal to”; /= means “is not equal to.”

```
expr = term {addopr term}
addopr = "+" | "-"
```

SS: If there is more than one **term**, they must all be of type integer.

DS: The first **term** is evaluated. Each subsequent **term** preceded by a + (respectively, -) is evaluated and its result added to (respectively, subtracted from) the overall result so far.

```
term = factor {multopr factor}
multopr = "*" | "/"
```

SS: If there is more than one **factor**, they must all be of type integer.

DS: The first **factor** is evaluated. Each subsequent factor preceded by a * (respectively, /) is evaluated and its result multiplied by (respectively, divided into) the overall result so far.

```
factor = [unopr] primary
unopr = "@" | "#"
```

SS: The **primary** after an @ must be of type character. The primary after a # must be of type integer.

DS: The **primary** is evaluated. If it is preceded by an @, the ASCII code of the character is yielded, and if it is preceded by a #, the character given by the ASCII code is yielded.