The Art *of* Compiler Design

THEORY AND PRACTICE

Thomas Pittman / James Peters

The Art of Compiler Design
Theory and Practice

Thomas Pittman

James Peters
University of Arkansas

PRENTICE HALL, Englewood Cliffs, NJ 07632

Library of Congress Cataloging-in-Publication Data

Pittman, Thomas.
The art of compiler design : theory and practice / Thomas Pittman, James Peters.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-048190-4
1. Compilers (Computer programs) I. Peters, James F. II. Title.
QA76.76.C65P57 1992
005.4'53--dc20 91-21591
CIP

Acquisitions editor: ***Tom McElwee***
Editorial/production supervision
and interior design: ***Richard DeLorenzo***
Copy editor: ***Brenda Melissoratos***
Cover design: ***Butler Udell Design***
Prepress buyer: ***Linda Behrens***
Manufacturing buyer: ***David Dickey***

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-048190-4

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

No useful modern complex information system ever evolved from the accumulation of chance events. The complexity of compilers as useful information systems can be understood in simple terms only because of careful design. Not only is the undesigned compiler nonfunctional, but compiler theory itself depends on a subtle relationship between linguistic principles originally developed for natural (human) languages and computers considered as finite-state automata. Understanding and exploiting this relationship requires a thorough grounding in the theoretical principles that underlie grammar theory, and a good grasp of their mechanical implementation. Thus, teaching compiler design involves also a complex information system that must be carefully designed to present the concepts in a coherent and logical sequence, so that the student finds compiler design both relevant and manageable.

This book is not an encyclopedic compendium of all possible ways to build all possible compilers, but a sequential introduction to the fundamental issues of compiler design in sufficient depth that the diligent student will be thoroughly equipped to construct practical and efficient compilers either by hand or using modern compiler generation tools, or some combination of the two. More importantly, the student will understand what is going on in the tools and why grammars must be composed in certain ways to achieve the intended results. This is the principle of design. Therefore, while most modern parser generators use bottom-up parsing techniques, we explore the more restrictive top-down parsing theory in considerable depth before advancing to a relatively brief (but complete) single chapter on bottom-up parsers. The goal in each chapter is always to instill an understanding of the concepts first, then to apply them in practical ways.

We believe *The Art of Compiler Design* stands out among comparable works in four important ways. First, it is rooted solidly and consistently in grammars. Beginning with the theoretical relationship between grammars and language recognizers, we continue throughout the entire book to apply the technology of grammars to all aspects of compiler design. The second distinction is the consistent and practical use of attribute grammars as a vehicle for compiler semantics. This uncompromising stand leads naturally to a compiler-compiler specified entirely in an attribute grammar that compiles itself; indeed, this is the focus of the final chapter. On the other hand, the third unique quality of this book is its very practical nature. Compiler *design* must be specified in attribute grammars, but compiler *construction* requires executable code, and every important

theoretical principle is illustrated in generous listings written in a real programming language, always showing the very natural relationship between the grammars and the machine code. Finally, our choice of Modula-2 as the programming language for illustrative code walks a narrow line between conceptual abstraction and concrete efficiency. Applying the optimizations taught in the later chapters can make programs written in Modula-2 more efficient at lower cost than comparable code in lower-level languages such as C.

This book may be used in a one-term beginning compiler course by concentrating on the first six or seven chapters, or the whole book may be spread over a full year for better coverage of the more advanced topics. The one-term course of study is suited for either a semester or quarter schedule; it has been optimized and classroom-tested to allow the steady progress of student projects culminating by the end of the term in a functional "Itty Bitty Modula" compiler that can be used in a final project to compile a *second* parser such at the pretty-printer or Tiny Basic interpreter outlined at the end of Chapter 6.

Some of the sections and problems are designed to be optional. While we firmly believe in a good theoretical basis for compiler design, there are some interesting mathematical side trails along the way which may be passed up if time or aptitude make them inappropriate. These have been marked with a little professor icon in the margin as you see here. Other sections are relevant to the major issues of compiler design, but the material is difficult to comprehend on the first-year level at which this book is aimed. These sections are identified with a little "Ex Calibur" icon in the margin as you see here. Similarly, there are some problems that are particularly challenging. The reader who tackles these problems will come away with agreater appreciation of the intricacies of compiler design, but a good understanding of the subject material does not require the extra time and effort that these problems take. They are also identified by the Ex Calibur icon in the margin.

While we can hardly mention all the people who contributed to this text, the first name that comes to mind is Brad Blaker, without whose encouragement and early assistance *The Art of Compiler Design* would never have happened. We also appreciate Bill Hankley, Austin Melton, and the patient students at Kansas State University for persevering through the early drafts, and Frank DeRemer for planting and nurturing many of the seminal ideas expressed here. The comments and suggestions from Thom Boyer, Dick Karpinski, Brian Kernighan, Marvin Zelkowitz, Wayne Citrin, Norman C. Hutchinson, Johnson M. Hart, Bernhard Weinberg, Will Gillett, and especially Dean Pittman, Chota, and Dave Schmidt were particularly helpful in making this into the book you are now reading.

Thomas Pittman
James Peters

Contents

Listings

Chapter 1
The Compiler Theory Landscape

Aims

- Survey the purpose of and approach to compiling.
- Introduce grammar concepts in language specification.
- Give an overview of a compiler structure.
- Introduce the basic data structures used by a compiler.
- Distinguish between lexical analysis and parsing.
- Survey the front and back ends of a compiler.

1.1 Introduction

Compiler design is one of the few areas of computer science where the abstract theory radically changed the way we write programs. The earliest compilers were largely written by ad-hoc "seat-of-the-pants" methods, using conventional programming techniques. The advent of grammar-driven parsers changed all that. We no longer see any real compiler that is not written first as a context-free grammar which is then mechanically translated into code.

This book is about modern compiler design, and so it is about grammars. Every part of a good compiler is related in some way or other to the grammars used to specify it. We show that the grammatical specification of the compiler *is* that compiler, written in a very high level language, and we show both how to write compilers in grammars and how to write grammar-compilers to compile the grammars into compilers. Grammar theory drives the design, so the designs are clean and easily implemented. The diligent reader can learn from this book how to write a complete compiler for a small but realistic programming language in a few days.

1.2 Languages and Translators

Like natural languages (English, French, Russian), computer languages define a way of structuring words into sentences for communicating information. A natural language communicates feelings of the heart, facts about the world, questions about those facts and feelings, and commands that should be followed by the listener or reader. A computer language is typically restricted to commands that are to be followed by the machine receiving them.

A natural language restricts the form of what can be said, but not what can be said. For example, it is meaningful in English to say, "Peter hit the ball" but not "ball Peter the hit." One is grammatically correct; the other is not. Similarly, we could say "Pièrre frappa la balle" in grammatically correct French. A bilingual reader would immediately recognize that the English and French sentences say the same thing — that is, they have the same meaning — but that would not necessarily be obvious to a person conversant in only one of those languages. The word "frappa" has no meaning to the English speaker, and the word "hit" means nothing to the French. Even if the dictionary meaning of the verbs were recognized, the grammars of the respective languages still define verb tense, which is indicated in the form of the words: both are in the past tense.

When an Englishman wishes to communicate to a Frenchman and neither knows the other's language, it is necessary to bring in a translator. In the natural world, a translator is a person who receives a message in one language and repeats that same message in some other language. A human translator from English to French would read, "Peter hit the ball" and write "Pièrre frappa la balle." If the translator happened upon the expression "ball Peter the hit," he would probably respond that it makes no sense. Because it has no meaning in English, it cannot be translated into a French sentence with any meaning.

A compiler is a computer program that acts like our human translator. It reads statements in one computer language, and if they make sense in that language, it translates them into statements with the same meaning in another computer language. There are rules defining what makes sense in each language, and the compiler applies these rules to determine if its input makes sense and to ensure that the output makes sense. A sequence of statements in a computer language is a program, and the compiler translates the program from one computer language (called the source language) into a program — that is, a sequence of statements — in another computer language (called the target language).

There are actually several kinds of computer languages and computer language translators. The simplest translator reads words in a simple computer language, and translates these words directly to the numbers that computers use for their instruction codes. This is called an *assembler*, and the source language is called *assembly language*. The name derives from the fact that most machine instructions are composed of several parts, and the assembly language uses a separate word or number for each part; the assembler assembles these parts into one numerical code. An assembler consists of little more than a table lookup routine, where each word of the source language is looked up in a table for its numerical equivalent, which is then output as part of the target language program. Assembly language generally gives the programmer precise and direct access to every capability of the computer hardware, but it is much harder to write correct programs in assembly language than in most other computer languages.

The term *compiler* is generally reserved for the more complex languages, where there is no immediate and direct relationship between the source language words and the target language. The target language for most compilers is usually the same machine language that is the target for assemblers — indeed, the purpose of computer language translators is to ease the process of creating programs in machine language — but most of the early compilers and even some modern compilers compile to assembly language and then let an assembler finish the translation to machine language. The source language for a compiler, however, is usually what we call a "high-level language" or HLL. High-level languages are characterized by resembling problem-solving notations rather than machine languages. For example, for business applications, Cobol ("COmmon Business Oriented Language") uses terminology easily understood by accountants and middle managers. Scientific problems are often stated in formulas for which Fortran ("FORmula TRANslator") is considered appropriate. Some programmers now favor a language with the more abstract structures of an HLL, but with all the low-level control offered by an assembler; for this purpose they use the language C (so named because it was the next language after an earlier language called B). Recent advances in programming methodology dictate a modular software design, a characteristic featured in Modula-2.

An *interpreter* is somewhat like a translator in that it reads a program in an HLL, but the translation is immediate, just as a human interpreter makes a verbal translation that is heard and understood immediately. Where a compiler will translate a computer program into machine code that executes at a later time, an interpreter actually executes the program as it is read. In one sense the interpreter never really completes the translation process; it is as if the human translator of our earlier example were to hear the command "Peter, hit the ball!" but instead of responding, "Pièrre, frappez la balle!" he just went and hit the ball himself. Because the interpreter does not have to be concerned with a target language, it can often process a line of source program much faster than a compiler. An interpreter must read its input program over and over to compute the results, but a compiler translates it only once. Compilers take longer to get the output from the first time a computer program is run, but subsequent runs are much faster than with the interpreter because no additional translation is needed.

Most of the focus of this book is on compiler design, but some of the exercises encompass interpreters also.

1.3 The Role of Grammars

One of the characteristics we study in a natural language such as English or French is its *grammar*. The grammar of a language defines the correct form for sentences in that language. For example, the English language might have some rules such as

sentence	→	noun-phrase verb noun-phrase
verb	→	"hit"
noun-phrase	→	article noun
	→	proper-name
article	→	"a" | "the"
noun	→	"ball" | "bat"
proper-name	→	"Peter"

This grammar says that a sentence can consist of a verb between two noun phrases. That is, the abstract concept of a sentence represented by the word "sentence" in the grammar may be rewritten as, or replaced by, the sequence of three abstract concepts, noun phrase, verb, and another noun phrase. A noun phrase can be a proper name like "Peter," or it can be an ordinary noun like "ball" with an article ("the" or "a"). The verb in our example is of course "hit." Similarly, in a computer language the grammar defines the correct form for sentences in that language by specifying how to rewrite the abstract concepts of the language as ever more concrete sequences of symbols. Each rewrite rule of the grammar is represented by an arrow connecting a word with one or more other words. Of course, what we mean by "sentence" is carefully defined in a computer language.

Starting with the word "sentence," this little grammar can generate not only the sentence "Peter hit the ball" but also "a ball hit Peter" and the somewhat nonsensical sentence "Peter hit Peter." When a grammar defines alternatives (either by means of multiple arrows, or else by the "or-bar" separator "|"), any one of the alternatives may be chosen arbitrarily in rewriting an occurrence of the name on the left of the arrow. The language generated by the grammar is the set of all the sentences that can be generated by successively choosing all possible alternatives in all combinations. This language has exactly 25 possible sentences.

A programming language is usually specified by two separate grammars, one to define the words of the language, the other to define how the words go together. A grammar could similarly be written to define the target language, and recent research has focused on building compilers automatically from the source and target grammars, but with mixed results. Therefore, we adhere to the more traditional compiler design methodology, using *attribute grammars* to define explicitly how the translation is to take place. Several grammars may be used to define a compiler, with each grammar specifying the functions of one component of the compiler.

The primary grammar is the *phrase-structure grammar*, and it specifies the central part of a compiler or interpreter, called the *parser*. The phrase-structure grammar specifies how the "words" of the computer language are allowed to fit together to form syntactically valid programs. "Parsing" is the natural-language term that describes the process of analyzing a sentence in that language according to its grammatical form; we use the term in exactly the same way with respect to computer languages. Our tiny English-language grammar is a phrase-structure grammar.

A secondary grammar is often used to specify the correct form or spelling of the "words" of the computer language. This is called the *lexical grammar*, from the Latin word for "word." The part of the compiler that analyzes the individual words of the input program is called the *scanner*. We might define a lexical grammar for English-language words something like this:

```
word  → letter word
      → letter
```

where we mean "letter" to stand for exactly one letter of the 26 letters of the alphabet at a time. Although this grammar generates a lot of nonsense words, it shows that a grammar need not be complex to generate a very large language. Indeed, the language of this grammar is infinite: as long as we choose the first alternative in adding single letters to the beginning of a word, the word can get as long as we like.

The translation process in a compiler is specified by values (called "attributes") and by attribute evaluation functions and assertions attached to the lexical and phrase-structure grammars. Other attribute grammars may be used to specify how the compiler can improve the speed or size of the compiled program. We could use assertions, for example, to prevent the nonsensical sentence "Peter hit Peter" by constraining the grammar to generate at most one proper name in a sentence.

This entire book focuses on the grammatical approach to compiler construction. It is our opinion that a grammar is a high-level language specification of a compiler — indeed, it *is* the compiler. In other words, all of the compiler design should go into writing the grammars that define the components of the language and its translation. That properly done, the rest of compiler design is mechanical and can be automated. The tools for automatically building compilers from grammars alone are fully described in this book, for these tools are themselves a compiler, and their design is an important part of the Art of Compiler Design.

1.4 Some Examples

One of the rewards of directing our attention to the construction of compilers as a logical extension of language specification is that we will be more likely to appreciate the clean design of grammar-specified and strongly-typed languages. Fortran was designed in a rather haphazard way to represent mathematical formulas and simple control structures. Without the grammatical specification to drive the language design, we find it excruciatingly difficult to write deterministic and fast scanners for Fortran compilers. Consider, for example, the following two lines, both legal in the original Fortran (note that spaces were insignificant in Fortran, and could be omitted entirely):

```
DO10K=1.9
DO10K=1,9
```

The first line is an assignment statement, assigning the real value `1.9` to the floating-point variable `DO10K`; the second is the beginning of a loop construct that ends on the statement labeled `10`, with the control variable `K`, which steps from one to nine. There is nothing to distinguish the line with statement label `10` as the terminator of a loop, so the compiler has no way to recognize whether or not the first is a miskeyed attempt to program the second (or the other way around). Indeed, it is reported that the first Venus probe was lost in a crash landing due to just such a programming error (a comma or period substituted for the other in such a way that it radically changed the meaning of the Fortran program controlling the space probe). Furthermore, in this example the compiler must examine all but the last character of this statement before it can even begin to consider what to do with the first — that is, whether this statement begins with the keyword `DO` or an identifier for a real variable.

Fortran is every language designer's favorite whipping boy; we mention only one other classic puzzle in the language, by way of example of the difficulties that could have been eliminated by a sound and unambiguous grammatical specification. In the following