COMPILING WITH C# AND JAVA

P.D. Terry, Rhodes University, 2004

CONTENTS

Preface

Acknowledgements

Chapter 1 - Translators and Languages

This chapter will teach you:

- why systems programs and translators play an important role in the use of high-level programming languages;
- the advantages and disadvantages of the use of high-level languages;
- the key concepts that underlie successful programming languages and programming paradigms;
- the benefits of using a formal description of a programming language.

1.1 Objectives
1.2 Systems programs and translators
1.3 The benefits of using high-level languages
1.4 The curse of complexity
1.5 Successful language design
1.6 The benefits of formal description

Chapter 2 - Translator Classification and Structure

This chapter will teach you:

- how to classify various types of translators, compilers and assemblers;
- the inner structure of typical translator programs;
- the phases involved in the translation process - in particular scanning, parsing and code generation;
- the mechanisms, advantages and disadvantages of using intermediate language interpretation as an alternative to native code generation;
- the concepts that underlie classic intermediate level systems such as the UCSD P-system, the JVM and the MicroSoft .NET Framework.

2.1 T-diagrams
2.2 Classes of translator
2.3 Linkers and loaders
2.4 Phases in translation
2.5 Multi-stage translators
2.6 Interpreters, interpretive compilers and emulators
2.7 The P-system and the JVM
2.8 JIT compilers and the .NET Framework

Chapter 3 - Compiler Development and Bootstrapping

This chapter will teach you:

- how a compiler for one language may be developed from a compiler for another language;
- how a compiler for one machine may be developed from a compiler for another machine;
- how a compiler may be developed to the point where it can compile itself

3.1 Using a high-level translator
3.2 Using a high-level host language
3.3 Bootstrapping
3.4 Self-compiling compilers
3.5 The half bootstrap
3.6 Bootstrapping from a portable interpretive compiler
3.7 A P-code assembler
3.8 The use of compiler generating tools

Chapter 4 - Stack Machines

This chapter will teach you:

- the fundamentals of simple machine languages and simple machine architecture - in particular as they apply to stack machines;
- how to develop assembler-level programs for a simple stack machine - the Parva Virtual Machine;
- how to emulate the PVM, how to enhance it and how to improve its efficiency and reliability;
- the principles of the design of the JVM and also of the conceptual stack machine that underlies the .NET CLR.

4.1 Simple machine architecture
4.2 ASSEMBLER languages
4.3 Addressing modes
4.4 The PVM - a simple stack machine
4.5 Programming the PVM
4.6 An emulator for the PVM
4.7 A minimal assembler for PVM code
4.8 Enhancing the efficiency of the emulator
4.9 Error handling in the PVM
4.10 Enhancing the instruction set of the PVM
4.11 Another stack machine - the JVM
4.12 The CLR - a conceptual stack machine

Chapter 5 - Language Specification

This chapter will teach you:

- what is meant by the syntax and semantics of a programming language;
- the terminology used in formal language description;
- various notations for specifying syntax, including regular expressions, syntax diagrams and Chomsky grammars;
- the use of BNF and EBNF notation for specifying the productions of a grammar;
- the principles behind formal descriptions of programming language semantics.

5.1 Syntax, semantics and pragmatics
5.2 Languages, symbols, alphabets and strings
5.3 Regular expressions
5.4 Grammars and productions
5.5 Classic BNF notation for productions
5.6 Simple examples
5.7 Phrase structure and lexical structure
5.8 ε-productions
5.9 Extensions to BNF
5.10 Syntax diagrams
5.11 Formal treatment of semantics

Chapter 6 - Development and Classification of Grammars

This chapter will teach you:

- the effective use of Chomsky grammars for describing familiar features of programming languages;
- how to identify pitfalls when designing grammars for programming languages;
- how to recognize and avoid ambiguity in the description of a language;
- what is meant by context-sensitive, context-free and regular systems of productions;
- the Chomsky hierarchy for classifying grammars according to the form their productions take.

6.1 Equivalent grammars
6.2 Case study - equivalent grammars for describing expressions
6.3 Some simple restrictions on grammars
6.4 Ambiguous grammars
6.5 The Chomsky hierarchy

Chapter 7 - Deterministic Top-Down Parsing

This chapter will teach you:

- how the formal definition of the syntax of a programming language can lead to algorithms for parsing programs written in that language;
- the restrictions that must be imposed upon grammars (and hence upon languages) that allow the use of simple top-down deterministic parsing algorithms;
- the Parva language (a simple imperative language based on C) used as a basis for future case studies.

7.1 Deterministic top-down parsing
7.2 Restrictions on grammars so as to allow LL(1) parsing
7.3 The effect of the LL(1) conditions on language design
7.4 Case study - Parva

Chapter 8 - Parser and Scanner Construction

This chapter will teach you:

- how the theory of the previous chapters can be used to construct top-down parsers for simple programming languages;
- how to construct scanners in an *ad hoc* way or from state diagrams;
- some special techniques for treating keywords and comments;
- the rudiments of bottom-up parser construction.

8.1 Construction of simple recursive descent parsers
8.2 Case study - a parser for assignment sequences
8.3 Other aspects of recursive descent parsing
8.4 Syntax error detection and recovery
8.5 Construction of simple scanners
8.6 Case study - a scanner for Boolean assignments
8.7 Keywords and literals
8.8 Comment handling
8.9 LR parsing
8.10 Automated construction of scanners and parsers

Chapter 9 - Syntax-directed Translation

This chapter will teach you:

- how to extend parsing algorithms to allow for semantic analysis to be carried out in parallel with syntactic analysis;
- how semantic actions can be performed by a translator while this analysis is being carried out or after it is completed;
- what is meant by an attribute grammar;
- notations suitable for attribute grammars.

9.1 Embedding semantic actions into syntax rules
9.2 Attribute grammars
9.3 Synthesized and inherited attributes
9.4 Classes of attribute grammars
9.5 Case study - a small database for a group of students

Chapter 10 - Using Coco/R: Overview

This chapter will teach you:

- the details of preparing attribute grammars for processing by Coco/R, a tool for the easy automated construction of scanners and recursive descent parsers.

10.1 Coco/R - a brief history
10.2 Installing and running Coco/R
10.3 Case study - a simple adding machine

10.4 Overall form of a Cocol description
10.5 Scanner specification
10.6 Parser specification
10.7 The driver program

Chapter 11 - Using Coco/R: Case Studies

This chapter will teach you how to use Coco/R

- to unravel the meaning of C declarations;
- to generate simple assembler code from simple expressions;
- to create abstract syntax trees for the generation of higher quality code from simple expressions;
- to analyze systems of EBNF productions;
- to construct a complete assembler for the PVM.

11.1 Case study - understanding C declarations
11.2 Case study - generating simple two-address code from expressions
11.3 Case study - generating two-address code from an AST
11.4 Case study - manipulating EBNF productions
11.5 Case study - a simple assembler for the PVM
11.6 Further projects

Chapter 12 - A Parva Compiler: the Front End

This chapter will teach you:

- how to use Coco/R to develop the front end of a compiler for simple imperative high-level languages - in particular the Parva language introduced in Chapter 7;
- how to carry out constraint analysis of Parva programs by constructing and interrogating a symbol table.

12.1 Overall compiler structure
12.2 File handling
12.3 Error reporting
12.4 Scanning and parsing
12.5 Syntax error recovery
12.6 Constraint analysis
12.7 Further projects

Chapter 13 - A Parva Compiler: the Back End

This chapter will teach you:

- the principles of code generation for a high-level language;
- how to develop the back end of a Parva compiler by utilizing a code generation interface from the front end developed in Chapter 12;
- how to develop a basic code generator for the PVM;
- how to extend and enhance the code generator to provide greater efficiency;
- the rudiments of the complexity of generating native machine code.

13.1 Extending the symbol table entries
13.2 The code generation interface
13.3 Using the code generation interface
13.4 Code generation for the PVM
13.5 Towards greater efficiency
13.6 Other aspects of code generation
13.7 Further projects

Chapter 14 - A Parva Compiler: Functions and Parameters

This chapter will teach you:

- the principles of storage management for more complex run-time systems;
- how to extend the basic Parva language to allow for multifunction programs;

- how to extend the PVM to support multifunction programs developed in extended Parva;
- the implications of extending Parva and the PVM to allow for mutually recursive functions, nested functions and garbage collection.

14.1 Source language extensions
14.2 Constraint analysis
14.3 Run-time storage management
14.4 Putting it all together
14.5 Mutually recursive functions
14.6 Nested functions
14.7 Heap management
14.8 Further projects

Chapter 15 - A C#Minor Compiler: the Front End

This chapter will teach you:

- how to use Coco/R to develop the front end of a compiler for the very simple object-oriented C#Minor language;
- fundamentals of the assembly languages for the JVM and CLR that can used as target languages for this compiler;
- how to design and construct ASTs for expressions and statements that will provide the interface between the front end and code generator of a C#Minor compiler.

15.1 The source language - C#Minor
15.2 The target language
15.3 The symbol table
15.4 Abstract syntax design
15.5 Abstract syntax tree construction
15.6 Parsing declarations
15.7 Further projects

Chapter 16 - A C#Minor Compiler: the Back End

This chapter will teach you:

- how to generate assembler code for the JVM or CLR by traversing the abstract syntax trees constructed by the compiler front end developed in Chapter 15.

16.1 Single-pass compiler strategy
16.2 Generating code for statements from an abstract syntax tree
16.3 Generating code for expressions from an abstract syntax tree
16.4 Low-level code generation - the CodeGen class
16.5 The OpCode class
16.6 Tracking the depth of stack
16.7 Generating the directives
16.8 Further projects

Appendix A - Assembler programmer's guide to the PVM, JVM and CLR

Appendix B - Library routines

Appendix C - Context-free grammars and I/O facilities for Parva and C#Minor

Appendix D - Software resources for this book

Bibliography

In the Resource Kit:

The programming language Parva (Level 1, Chapter 13)

The programming language Parva (Level 2, Chapter 14)

The programming language C#Minor