University of Glasgow | School of Computing Science

Implementation and Visualisation of the Contextual Analysis and Code Generation Phases of a Compiler

David Robertson

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — January 1, 2000

Abstract

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ____________________

Contents

Chapter 1

Introduction

1.1 Motivation

With increasing technological advances and furthering levels of software abstraction, some may consider compilation to be a slightly esoteric subject; in that, the specific knowledge acquired can be directly applied only in a small number of highly specialized industries. Yet, remaining as a foundation of a computer science curriculum at many schools and universities is the art of compiler construction, behaviour and optimization.

Niklaus Wirth, the creator of the Pascal programming language and renowned lecturer of compiler design states, “*knowledge about system surfaces alone is insufficient in computer science; what is needed is an understanding of contents*” and that, “*[compilers] constitute the bridge between software and hardware*”. Wirth’s view is one that I share, in the respect that the most successful computer scientists must have more than a superficial understanding of which approaches to follow in a given situation; they must understand how various components interact and why they behave the way they do, as this is the only way of making deeply informed technological decisions.

Compilation can often be a tricky field to teach. Most compilation procedures require the construction of auxiliary data structures, commonly trees, which must be traversed in a specific manner in order to both validate the source code and create some useful output, usually an executable program. In order to illustrate this concept to students, the educator is typically restricted to creating a “slideshow”, using an application such as Microsoft Powerpoint, in which each slide shows a distinct step of the traversal. Creating a slideshow to demonstrate these concepts is not only an onerous task on behalf of the educator, but it is more importantly restricted to showing strictly pre-determined examples.

1.2 Compilation

1.2.1 Overview

The term “high-level” language is used to refer to a programming language that provides significant levels of abstraction. These languages often allow the programmer to hide or automate several aspects of a computer system, such as memory management or garbage collection. Whilst these languages are much easier to use than their low-level counterparts, it is usually not possible to execute source code written in a high-level language directly on a computer. Examples of high-level languages are Java, C++ and Haskell.

Conversely, a “low-level” language provides little to no abstraction. Statements written in a low-level language often map very closely to processor instructions. Despite the fact that these languages are much more difficult to use directly, and in some cases are not in a human-readable format, source code written in a low-level language can usually be executed directly on a computer. Examples of low-level languages are assembly language, object code and machine code.

Compilation is the process of translating high-level code into low-level code. The most common case is to convert a program whose source code is written in some programming language, into an executable program. This compilation process can usually be decomposed into three distinct phases: syntactic analysis, contextual analysis and code generation.

1.2.2 Syntactic Analysis

During syntactic analysis the source program is parsed to check whether it is “well-formed”, in accordance to the language’s syntax. Syntactic analysis can itself be broken down into two further phases: lexing and parsing. A lexer takes a source program as input, and breaks it down into a stream of “tokens”, this stream is then passed to the parser which converts the token stream into an abstract syntax tree (AST) using some parsing algorithm. The parsing algorithm [used throughout] is recursive-descent parsing. If any errors are encountered during syntactic analysis, the compilation process is halted.

1.2.3 Contextual Analysis

Upon successful completion of syntactic analysis, the AST is traversed or “walked” by the contextual analyser. The contextual analyser will check whether the program represented by the AST conforms to the source language’s scope and type rules. Contextual analysis can be broken down into two further phases: scope checking and type checking. Scope checking ensures that every variable used in the program has been declared. Type checking ensures that every operation has operands of the expected type. If any errors are encountered during contextual analysis, the compilation process is halted.

1.2.4 Code Generation

Upon successful completion of contextual analysis, the code generator translates the parsed program into a lower level language, such as assembly language or object code. Code generation can be broken down into two further phases (in the case of stack-based VMs): address allocation and code selection. Address allocation decides the representation and address of each variable in the source program. Code selection selects and generates the object code. Upon successful completion of this phase, we should have an executable program.

1.3 Background

1.3.1 ANTLR

ANTLR (Another Tool For Language Recognition) is a popular compiler generation tool. Given a grammar written in ANTLR notation (which is similar to EBNF notation), ANTLR can automatically generate a lexer and a recursive-descent parser. ANTLR also generates a visitor interface, which is the foundation for implementing a contextual analyser and a code generator.

1.3.2 The Fun Programming Language

Included in Niklaus Wirth's 1975 book "Algorithms + Data Structures = Programs", was a language written entirely in Pascal named "PL/0". PL/0 was intended as a small educational programming language used to teach the concepts of compiler construction. The language contains very primitive constructs and limited operations. Similarly to PL/0, "Fun" is a simple imperative language, developed at Glasgow University by David Watt and Simon Gay. Its purpose is to illustrate various general aspects of programming languages, including the construction of an elementary compiler. The language is provided as a supplementary aid during the delivery of the level 3 Computer Science course, "Programming Languages", at Glasgow University.

1.3.3 A subsection

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox [2] jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

Chapter 2

The Fox and Dog

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

2.1 The Fox Jumps Over

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over [1] the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

2.2 The Lazy Dog

The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

The quick brown fox jumped over the lazy dog. The quick brown fox [3] jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog. The quick brown fox jumped over the lazy dog.

Appendices

Appendix A

Running the Programs

An example of running from the command line is as follows:

```
> java MaxClique BBMC1 brock200_1.clq 14400
```

This will apply $BBMC$ with $style = 1$ to the first brock200 DIMACS instance allowing 14400 seconds of cpu time.

Appendix B

Generating Random Graphs

We generate Erdós-Rënyi random graphs $G(n,p)$ where n is the number of vertices and each edge is included in the graph with probability p independent from every other edge. It produces a random graph in DIMACS format with vertices numbered 1 to n inclusive. It can be run from the command line as follows to produce a clq file

```
> java RandomGraph 100 0.9 > 100-90-00.clq
```

Bibliography

[1] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings IJCAI'91*, pages 331–337, 1991.

[2] Torsten Fahle. Simple and Fast: Improving a Branch-and-Bound Algorithm for Maximum Clique. In *Proceedings ESA 2002, LNCS 2461*, pages 485–498, 2002.

[3] Brian Hayes. Can't get no satisfaction. *American Scientist*, 85:108–112, 1997.