University of Glasgow | School of Computing Science

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

With increasing technological advances and furthering levels of software abstraction, some may consider compilation to be a slightly esoteric subject; in that, broad knowledge of compilation theory is unnecessary in a modern environment and required only in a small number of highly specialised industries. Yet, remaining as a cornerstone of a computer science curriculum at many schools and universities is the art of compiler construction, behaviour and optimisation.

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*". Wirth's view is one that many 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.

1.1 Motivation

Compilation can often be a challenging field to teach effectively. Most compilation procedures involve the generation and traversal of complex data structures. These aspects can often be difficult for students to understand as the data structures can be indefinitely large and the traversals situationally specific.

In order to explain these concepts to students, an educator will typically try to illustrate the process. The educator is usually restricted to creating a "slideshow", using an application such as Microsoft PowerPoint, in which each slide shows a distinct step of the traversal.

Whilst using a slideshow is currently the only practical way to demonstrate these concepts, it has two main drawbacks. Firstly, to create animations in this way is an arduous task for the educator, realistically meaning that the animation must remain short. Secondly, and more importantly, the educator is restricted to showing only pre-determined examples. Since there is effectively an infinite number of ways the compilation process may occur depending on the input, pre-determined examples are almost guaranteed to omit certain details, making it difficult for students to achieve a generalised understanding.

1.2 Aim

The aim of this project is to provide a web application in which users will be able to visualise the contextual analysis and code generation phases of the Fun compiler. The visualisation will be in the form of a representation of an abstract syntax tree (AST). The mechanics of each phase will be illustrated by "jumps" over the AST, demonstrating the traversal that is internally taking place within the compiler. Chapter 2 introduces these concepts in considerably more detail. The application should:

- Allow a user to input and submit any syntactically valid program written in the Fun language.
- Visualise the contextual analysis phase of a program's compilation.
- Visualise the code generation phase of a program's compilation.
- Display any relevant details during the compilation, including: address/type tables, code templates, object code and explanatory messages.
- Allow the visualisation to be "played" continuously or step-wise, backwards and forwards.

This application will act as a teaching tool, equally useful to educators and students alike. Students are free to use the tool outside of school/university hours in order to further their own learning. Since any arbitrary Fun program can be compiled, the restriction to educators of showing only pre-determined examples is removed. Additionally, the level of automated analysis attainable from the application is considerably greater than anything currently possible by present techniques. It will hopefully provide a better means for those looking to learn but also remove some of the struggle taken on by educators in teaching the topic.

1.3 Outline

The rest of this report is organised as follows. Chapter 2 takes a general look at compilation and its constituent phases. Chapter 3 discusses related work in the area and considers relevant tools. Chapter 4 looks at requirements gathering and elicitation. Chapter 5 discusses how the requirements were used to build an initial design of the system. Chapter 6 covers in detail the implementation of the application.

Chapter 2

Background

This chapter aims to briefly introduce the reader to the main tools and concepts used throughout this paper. This includes a small overview of syntax trees, the various phases of compilation, the Fun programming language and the tool ANTLR.

2.1 Syntax Trees

A syntax tree is simply a hierarchical representation of a source program; with global statements towards the root, and more deeply nested statements towards the leaves. We typically consider two types of syntax tree: the concrete syntax tree (often called a parse tree) and the abstract syntax tree (AST). A parse tree contains an exact representation of the input, retaining all information, including white-space, brackets, etc. Conversely, an abstract syntax tree is a smaller, more concise representation of the parse tree. An AST usually ignores words and characters such as white-space, brackets and other redundant details which are derivable from the shape of the tree.

During compilation, the compiler will traverse one or sometimes both types of tree, depending on the language implementation. The compiler usually visits each node in the tree in a "depth-first" manner, perhaps with some small variations. These traversals constitute the contextual analysis and code generation phases of a compiler, which validate certain aspects of the input and produce the output of the compilation.

From an visual point of view, ASTs are considerably easier to read and understand. Any information that is lost from the conversion of a parse tree to an AST is purely semantic and does not effect how the tree is evaluated. Consequently, when attempting to demonstrate the data structures built during compilation to students, it is much more productive to show an AST, as opposed to a parse tree. See chapter 3 for an illustrative example that displays the visual benefits of ASTs over parse trees. Beyond visualisation, it's worth noting that there can also be large computational advantages of using an AST within the compiler's implementation as they are usually much easier to manipulate and require less memory; however, the compiler of the Fun language does not happen to take advantage of this.

2.2 Compilation Phases

Compilation is the process of automatically 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:

a) *Syntactic Analysis*

b) *Contextual Analysis*

c) *Code Generation*

If either syntactic analysis or contextual analysis encounters an error (as specified by the language) during its execution, that particular phase completes, the remainder of the compilation process is halted and the errors are reported to the programmer. See figure 2.1 for a diagram of a typical compilation pipeline.

2.2.1 Syntactic Analysis

During syntactic analysis the source program is inspected to verify whether it is well-formed in accordance to the language's syntax. Syntactic analysis can be broken down into *lexing* and *parsing*.

Lexing is the process of breaking down an input program into a stream of tokens. Parsing converts this stream into an AST using a parsing algorithm. The parsing algorithm used in the Fun compiler is recursive-descent parsing.

2.2.2 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 *scope checking* and *type checking*.

Scope checking ensures that every variable used in the program has been previously declared. Type checking ensures that every operation has operands of the expected type.

2.2.3 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 *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 final phase, the compiler will have often produced an executable program.

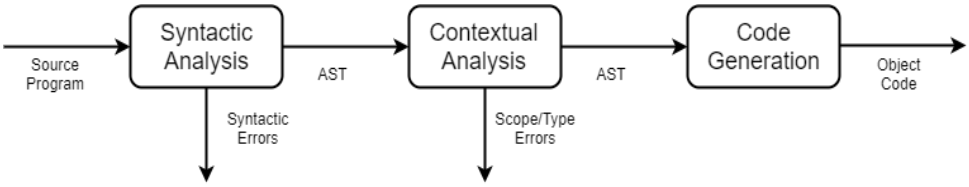


Figure 2.1: Compilation Pipeline

2.3 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 built using ANTLR, developed at Glasgow University by David Watt and later extended by 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.

The Fun compiler will be used to visualise compilation within the web application. Whilst Fun may differ significantly to other programming languages, particularly in its complexity; it is not the case that the concepts illustrated are exclusive to the Fun language or the Fun compiler. Fun is sufficiently generic that the core concepts can be explained in an easily understandable format (due to the simplicity of the language), which then facilitates learners in applying the same logic to more complex constructs in other languages.

Chapter 3

Related Work

Whilst compiler visualisation is a considerably novel area of research and development, attempts to visualise various computing algorithms date as far back as the 1980s. The vast majority of work in this field focuses on animating sorting algorithms, such as quick sort and merge sort. Understandably, many would consider compilation to be a much more involved process than a typical sorting algorithm; however, compilation is simply just an algorithm, or a sequence of algorithms. Consequently, we believe that the lessons learnt and guidelines imparted in regards to effective algorithm animation apply the same to this project.

This chapter considers existing products in the area of compiler visualisation and discusses research done in the field of visualising computing algorithms in general, beyond just compilation.

3.1 History

3.2 Effectiveness of Algorithm Animation

Despite the fact that compiler visualisation is a considerably novel area of research and development, attempts to visualise computing algorithms for educational purposes started long ago. including work from Bentley and Kernighan.

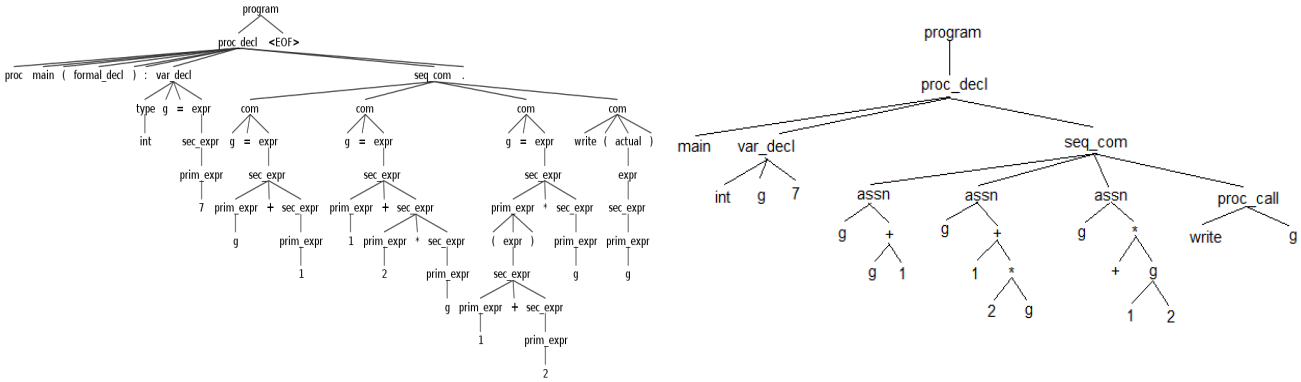
The outcome of studies which aim to measure the effectiveness of animating algorithms as a learning aide have been disappointing over the years. Whilst students have certainly found benefits in observing these animations compared to static materials, such as textbooks, the result has not been as significant as we might intuitively believe.

3.3 Existing Products

Interestingly, compiler visualisation is a considerably novel area of research and development. Very few existing tools provide any illustration of the various components of the compilation process and essentially no tools exist that provide any analysis or step-by-step explanation of the process.

ANTLR, the compiler generation software used to build the Fun language, does provide a tool for visualising the parse tree generated from an input program, as seen in figure 3.1a, which represents the parse tree of a small

program written in Fun. As previously discussed, a parse tree is much harder to interpret than its AST counterpart, a potential version of which is illustrated in figure 3.1b; unfortunately, ANTLR currently provides not out of the box support to build trees in this way. Whilst this tool may serve to provide a preliminary understanding of how a parse tree may be formed for a particular program, we can immediately observe how large and difficult to read figure 3.1a is, in comparison to figure 3.1b. There are many unnecessary tokens, such as EOFs, brackets and colons which serve only to obfuscate the diagram. The tree also contains many paths consisting of multiple unary branch nodes, which could easily be collapsed into a single edge. Most of all, this tool provides nothing more than a simple diagram, it gives no indication of how this tree would be traversed during compilation, nor does it reveal any details of the internal workings of each node.



(a) Parse Tree/Concrete Syntax Tree

(b) Abstract Syntax Tree

Figure 3.1: The ANTLR generated parse tree and the theoretical AST of the same Fun program

Chapter 4

Requirements

4.1 Methodology

After establishing that a product is worthwhile to build in the first place, most requirement elicitation approaches involve a repetitive cycle of expanding or reducing an initially small list of desiderata. After an initial interview with Simon Gay, the current lecturer of the *Programming Languages* course and proposer of this project, it was clear that whilst the project was inherently complex, the set of functional requirements was simple, well-defined and unlikely to change significantly in the future. For this reason, it was determined that extensive requirement gathering techniques, such as questionnaires or focus groups, were ultimately unnecessary; and all additional requirements were to be established through further interviews with Simon Gay.

4.2 User Stories

User stories are short and simple descriptions of a potential feature, told from the perspective of a potential user. User stories typically follow the template of:

*As a **user type**, I want to **achieve some goal**, so that **justification.***

User stories are a core component of the agile software development approach. They provide a means of considering the possible features that various different types of user may want in order to build a fuller set of functional requirements. User stories can also provide the basis for task estimation and prioritisation; however, since this project is not being developed by a team, these aspects won't be considered aside from a high-level prioritisation of end functional requirements.

- As a user, I want to read details about the Fun language, so that I can write valid Fun programs as input and better understand the compilation animations.
- As a user, I want to be able to input any Fun program, so that I can learn about the compilation process in the general case, not just for specific examples.
- As a user, I want to be able to view the animation of the contextual analysis phase of my program, so that I can understand how the compiler carries out this task.
- As a user, I want to be able to view the animation of the code-generation phase of my program, so that I can understand how the compiler carries out this task.

- As a user, I want to be able to play different sections of the compilation animation independently (i.e., contextual analysis or code generation), so that I can focus my learning on specific areas.
- As a user, I want to be able to replay an animation, so that I can review any details I missed/misunderstood.
- As a user, I want to be able to step through the animation at my own pace, so I can easier understand what is happening during the animation.

4.3 Functional Requirements

After conducting several interviews with Simon Gay and analysing the above user stories, a formal list of functional requirements was created. Functional requirements are intended to capture a specific function of a system. The *MoSCoW method* was used as a prioritisation technique for the following requirements. This method is another commonly used aspect of agile development, and involves partitioning requirements into four categories: *Must have*, *Should have*, *Could have* or *Would have*.

4.3.1 Must Have

- Allow users to view the AST for pre-defined Fun programs.
  - At the very least, users should be able to choose from a small list of pre-written Fun programs and view the corresponding AST.

4.3.2 Should Have

- Allow users to view a simple continuous animation that demonstrates how the AST would be traversed during contextual analysis and code generation.
  - The animation cannot be paused, reversed, or moved through step by step.
- At the end of the animation, display some basic results of the compilation, including object code and address/type tables.
  - These details would only be published at the end of the animation, not during.
- Display information that explains how the Fun language works.
  - This would involve effectively embedding the Fun specification somewhere within the application.

4.3.3 Could Have

- Allow users more control over the animation, including pausing, reversing, and step-wise movements - backwards and forwards.
- Allow users to input any arbitrary Fun program and view the corresponding animation.
  - The user is no longer restricted to using pre-defined example programs.
- Display results of the animation as they occur during the compilation.
  - For example, populate the type table as each variable is declared during the animation.

- Display more in-depth analytical and informational details during the animation.
  - This includes code templates and explanatory messages of the internal workings of each node as it is visited.

4.3.4 Would Have

- Execute and display the results of the generated object code.

4.4 Non-functional Requirements

In contrast to functional requirements that detail specific behaviours or a system, non-functional requirements measure an overall properties of a system. Non-functional requirements often consider areas such as security, usability and extensibility; overall utilities of a system.

- The application must work on all modern browsers.
- The application must be able to interact efficiently with a Java-based application (the Fun compiler).
- The application must be responsive, at least to a tablet level.
- The application must ensure no malicious code can be executed.

Appendices