University of Glasgow | School of Computing Science

Animation 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*"[6]. 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 Aims

This project aims to alleviate or solve many of the issues currently faced in attempts to create effective visualisations of various compilation concepts. The project will provide a web application in which users will be able to animate the contextual analysis and code generation phases of the Fun compiler. The animation 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 3 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. Addtionally, since any arbitrary Fun program can be compiled, the restriction to educators of showing only pre-determined examples is removed. Finally, 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...

Chapter 2

Background & Related Work

Despite the animation of compilers being a considerably novel area of research and development, attempts to visualise computing algorithms date as far back as the 1980s[1]. The vast majority of work in the field up to now has been focused on the animation of complex, yet small and well-defined algorithms, most notably sorting algorithms or tree traversals. Indeed, compilation is certainly not small nor particularly well-defined; however, many aspects of typical algorithm animations, such as tree traversals, can be found in abundance within a potential compiler animation. Ultimately, compilation itself is just an algorithm, and consequently it would seem logical to assume that any lessons learnt during the development and evaluation of existing algorithm animation software should be equally applicable to the area of compiler animation.

The remainder of this chapter considers research done which attempts to evaluate the effectiveness of algorithm animation from an educational perspective. We then explore and critique some modern examples of web-based algorithm animation tools. Finally, we discuss how we might use the results of prior evaluations along with our analysis of existing products to influence our design of a compiler animator.

2.1 Effectiveness of Algorithm Animation

One of the earliest algorithm animation systems was developed by Bentley and Kernighan in 1987[1]. The system enabled users to annotate sections of an algorithm which were later processed by an interpreter to create a sequence of still pictures, an example of which is shown in Figure 2.1. In the very first line of the system's user manual Bentley and Kernighan confidently state, "*Dynamic displays are better than static displays for giving insight into the behaviour of dynamic systems*". This belief of Bentley and Kernighan is one that many of us would intuitively believe. The intuition being that when attempting to understand any multi-step process, an animation which displays each step of that process is more effective than a single static diagram, or a section of explanatory text. However, it is important to consider whether this belief has statistical backing or whether it is simply an assumption. Certainly, in 1987 algorithm animation itself was still in its infancy and no studies had been carried out that provided the empirical evidence to support this intuition.

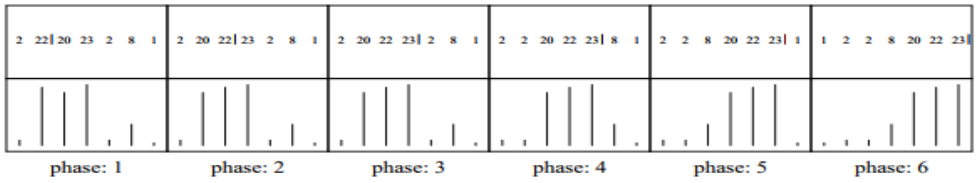


Figure 2.1: A sequence of stills from an insertion sort algorithm

Six years later in 1993 Stasko, Badre and Lewis were amongst the first to considered whether algorithm animations assisted learning as much as we might think[2]. Stasko, Badre and Lewis carried out a study which involved attempting to teach students the concept of a "pairing heap", with one group using textual descriptions of the algorithm, the others using an animation. Stasko then repeated a similar experiment in 1999 with Byrne and Catrambone which more so focused on the benefits of animations over static visualisations [3]. In both studies the researchers found the results to be disappointing. They found that whilst students in the animation group did perform better than their textual or static counterparts, the improvement was not statistically significant.

Despite computer graphic capabilities improving significantly since the 1990s, other more recent studies have all shown the same results. The general consensus being that whilst animations do provide a small benefit, it is certainly not as large as our intuition would lead us to believe. However, that is not to say that algorithm animations are without use. Stasko, Badre and Lewis suggest that algorithm animations are not particular effective when students are trying to learn a concept for the first time, but is likely to be much more suitable when students are looking to refine their understanding of a particular notion. The theory is that students should ideally learn the primitive concepts of the algorithm using conventional methods initially, then transition to using animations when looking to clarify and solidify their understanding of certain aspects.

Stasko, Badre and Lewis also reveal a list of guidelines they believe to be effective advice when building algorithm animations, some of which are summarised below:

- The animation should be augmented with textual descriptions.
- The animation should include rewind-replay capabilities.
- Students should be able to build the animation themselves.

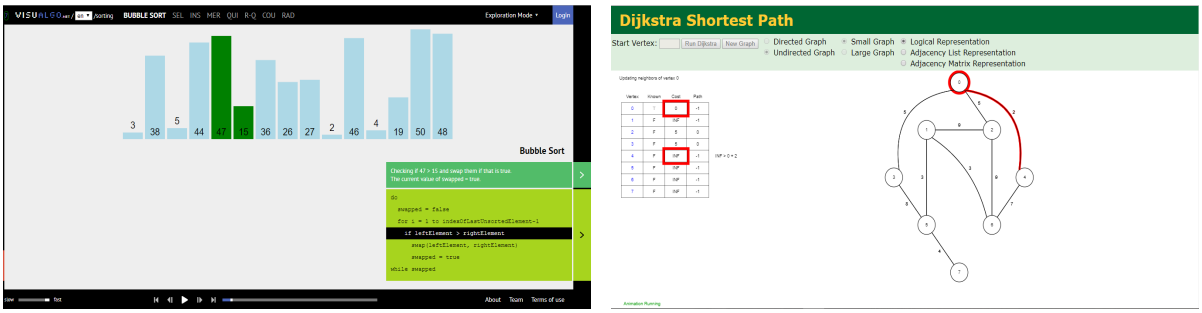
These guidelines suggest that algorithm animations require accompanying messages that explain the logic of the algorithm at each step. Also, the animation should be interactive in order to engage students in "active learning" over "passive learning". This interactivity would include intricate controls over the playback of the animation and the ability to modify the input of the algorithm.

2.2 Existing Products

Currently, very few existing tools provide even static illustrations of compilation components (such as syntax trees) and virtually no products exist that create animations of the compilation process. There are however, an abundance of web applications that animate more straightforward algorithms, such as sorting or searching algorithms.

One of the most popular and well implemented is VisuAlgo[5]. VisuAlgo provides an interface for animating various sorting algorithms, including bubble sort, selection sort, etc. As shown in Figure 2.2a, Visualgo implements many of the guidelines we previously listed. It provides impressive playback controls, the ability to choose the input of the algorithm and displays in-depth descriptions of the current step in the algorithm, in both plain English and pseudo-code.

Moving away from pure sorting algorithms, the University of San Francisco developed a small web application which animates Dijkstra's algorithm[4], as shown in Figure 2.2b. Dijkstra's algorithm is arguably more complicated than many sorting algorithms, involving the need to visualise tables and graphs. However, we see that at the expense of this complexity, the application has sacrificed several usability aspects in comparison to VisuAlgo. You can not define your own input and you can not pause or rewind the animation. Additionally, whilst some very primitive details are displayed next to the table at each stage, they are far from informative explanations.



(a) VisuAlgo (b) The University of San Francisco

Figure 2.2: Animations of different algorithms from different applications

2.3 Discussion

Despite studies showing that algorithm animation is not as effective as we might have initially believed, it would seem that the circumstances that often lead to surprisingly poor performance are unlikely to effect the web application we aim to build, mainly due to the environment in which it is targeted for use. The studies found that algorithm animation was less effective in novice students who were learning the algorithms for the first time; however, the intended users of this application will already be studying some compilation-related course, where the tool will act as a secondary learning resource. The users will have ideally been taught the concepts using conventional methods to begin with, then the educator or the students can utilise the application as a means to refine and clarify understanding.

When considering the previous guidelines proposed by Stasko, Badre and Lewis and how they might affect our application, it appears the system should:

- Provide supplementary textual explanations that rationalise and justify the logic of the compiler as it visits each node in the AST.
- Include rewind-replay functionality that allows the compilation animation to be restarted, or played step-wise, backwards and forwards.
- Ensure users can create the animation themselves by allowing any arbitrary Fun program as input to the compiler.

After analysing some examples of current products that are available in the area of algorithm animation, such as VisuAlgo and the University of San Francisco's Dijkstra's algorithm animator, it seems there is a trade-off between complexity and usability. In order to animate a considerably more complex algorithm, the University of San Francisco sacrifices much of the visual support and playback control that Visualgo is able to provide in order to animate a more convoluted algorithm. Our web application desires to visualise and animate an algorithm that is certainly more complex and volatile than even Dijkstra's algorithm, but still provide the level of usability that a website like VisuAlgo manges to offer. This is necessary because an in-depth compilation animation requires more than just a simple traversal over a tree; it requires the display of multiple tables, code templates, generated code and highly input-dependent explanatory messages. Yet, as we've seen from previous studies, if we are to create an animation that is as effective as possible from a learning perspective, the usability and interactivity aspects such as playback control that are embedded within the guidelines in these studies are crucially required.

Chapter 3

Concepts

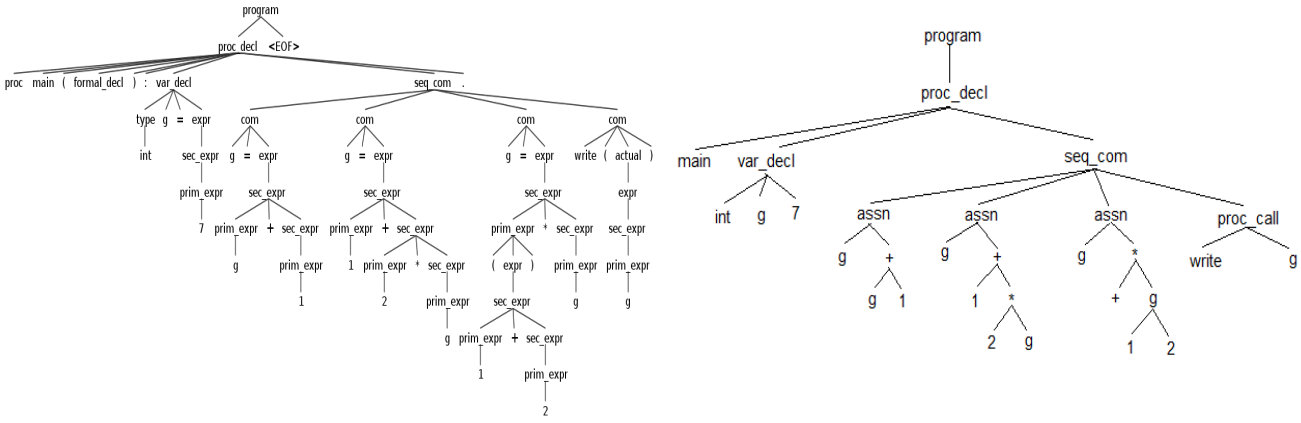
This chapter aims to briefly introduce the reader to the main tools and concepts used throughout this paper. This includes a small overview of syntax trees, the various phases of compilation, the Fun programming language and the tool ANTLR.

3.1 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. Figure 3.1 illustrates the visual differences between a parse tree and an AST of the same hypothetical Fun 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 parse tree also contains many paths consisting of multiple unary branch nodes, which could easily be collapsed into a single edge. 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.



(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

3.2 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 3.2 for a diagram of a typical compilation pipeline.

3.2.1 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.

3.2.2 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.

3.2.3 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.

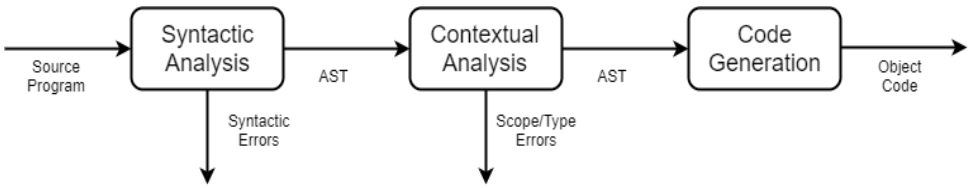


Figure 3.2: Compilation Pipeline

3.3 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 4

Requirements

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, 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 and using any insight gained from background research.

The remainder of this chapter lists the web application's user stories, prioritised functional requirements and non-functional requirements.

4.1 User Stories

User stories are short and simple descriptions of a 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 desire in order to build a fuller set of functional requirements. When we talk about different "types" of user we are often referring to groups of users who have different permissions or entirely different objectives from other groups, i.e., admin and non-admin. However, these distinctions aren't relevant or required within our web application, and therefore all users are referred to simply as "user".

- 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 more easily understand what is happening during the animation.

4.2 Functional Requirements

After conducting several interviews with Simon Gay, carrying out background research and analysing the above user stories, a formal list of functional requirements were 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*.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 Would Have

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

4.3 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.

Chapter 5

Design

Placeholder.

5.1 Overview

Ultimately, use of the FunCompiler requires only two main user actions. One is inputting a Fun program, the other is controlling the playback of the resulting compiler animation. The rest of the application is dedicated to displaying visualisations, animations and analytical information. Given the concise functionality of the FunCompiler, it was deemed unnecessary to provide a large number of individual web-pages. Additionally, due to the nature of the application and the inherent link between the input, the animation, and the analytics; it would seem almost necessary that these details are available within the same view.

probs need a bit here to say i therefore put everything in one view.

Consequently, it was decided that the FunCompiler would follow the principles of single-page applications (SPAs). SPAs aim to significantly reduce the number of page refreshes websites undergo when the user requests new information or desires to see a different view (i.e., clicks a button). Treating the FunCompiler as an SPA would mean that when users, for example, submit a Fun program, the application will dynamically load the animation into view, without requiring a page refresh. The advantage of SPAs is that for small websites, you can create a much faster workflow which appears seamless. SPAs do have downsides, usually in terms of forcing too much processing on the client when websites grow large, and search engine optimisation. Fortunately, the FunCompiler suffers from neither of these problems.

5.2 Application Interface

Appendices

Bibliography

[1] Jon L. Bentley and Brian W. Kernighan. *A System for Algorithm Animation Tutorial and User Manual*. 1987.

[2] Clayton Lewis John Stasko, Albert Badre. *Do Algorithm Animations Assist Learning? An empirical Study and Analysis*. 1993.

[3] John T. Stasko Michael D. Bryne, Richard Catrambone. *Evaluating Animations as Student Aids in Learning Computer Algorithms*. 1999.

[4] The Univeristy of San Francisco. Dijkstra's algorithm animation. `https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html`. [Online; accessed 03-February-2018].

[5] VisuAlgo. Sorting algorithm animations. `https://www.visualgo.net/en/sorting`. [Online; accessed 03-February-2018].

[6] Niklaus Wirth. *Theory and Techniques of Compiler Construction*. 2005.