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.

Compilation theory is a foundational notion of computer science, upon which a larger syllabus can be developed. To students, the benefits of learning about compilation are perhaps not in its direct application, but in its ability to act as the stepping stone that facilitates a more comprehensive grasp of other concepts; concepts which themselves are grounded in compilation theory.

When questioned on the necessity of teaching compilation, Niklaus Wirth, the creator of the *Pascal* programming language and renowned lecturer of compiler design remarks, "*knowledge about system surfaces alone is insufficient in computer science; what is needed is an understanding of contents*" [7]. 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. True understanding of the building blocks of computer science yields a better appreciation of the scientific and technical concepts within the field, as well as providing the only effective means 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 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 broad and complete 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 incorporate a representation of an ***abstract syntax tree*** (AST).

Fun is a simple educational programming language. The compiler we will use within the web application will be the Fun compiler. An AST is a data structure that represents the hierarchical structure of an input program. Contextual analysis and code generation are simply two different stages of compilation which aim to verify certain parts of the input program and construct an output program. The behaviour of these stages is expressed by traversals over an AST. Chapter 3 introduces these concepts in considerably more detail.

Within the application, the mechanics of each phase will be illustrated by "jumps" over the AST, demonstrating the traversal that is internally taking place within the compiler. 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 animation to be played, paused and moved 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. Additionally, 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 [2]. 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 (in that the behaviour of the compiler can vary significantly depending on the implementation); 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 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 [2]. 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 5.1. 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.

Six years later in 1993 Stasko, Badre and Lewis were amongst the first to consider whether algorithm animations assisted learning as much as we might think [3]. 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

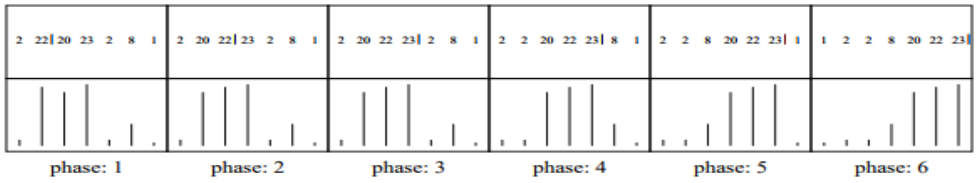


Figure 2.1: A sequence of stills from an insertion sort algorithm

and Catrambone in which one group used an animation and the others used static visualisations (such as diagrams from a textbook), instead of textual descriptions like in the previous experiment [4]. In both studies the researchers found the results to be disappointing. They found that whilst students in the animation group did perform moderately better than their textual or diagrammatic 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:

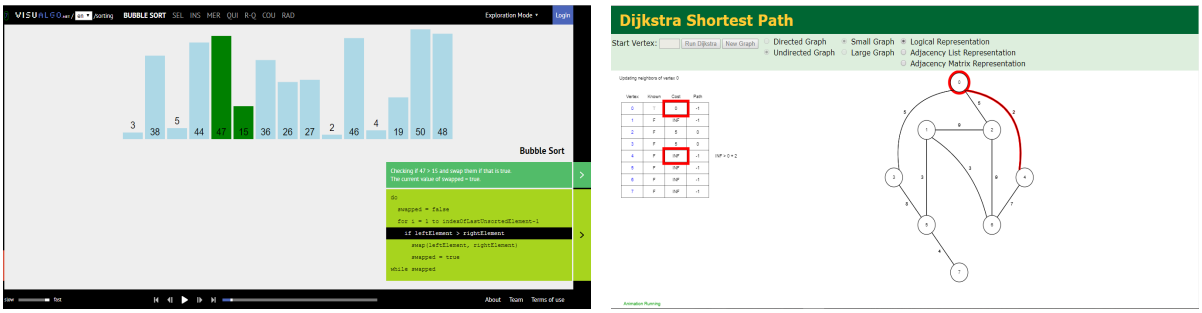
- The animation should be augmented with textual descriptions.
- The animation should include rewind-replay capabilities.
- Students should be able to build the animation themselves.

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 tools exist that create animations of the compilation process as a whole. 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 [6]. 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.



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

Figure 2.2: Animations of different algorithms from different applications

Moving away from pure sorting algorithms, the University of San Francisco developed a small web application which animates Dijkstra's algorithm [5], 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.

2.3 Discussion

Despite studies showing that algorithm animation is not as effective as we might have initially believed, it would seem that the surprisingly poor performance is due to specific circumstances which could alleviated if the environment in which an algorithm animation tool was targeted for use was selected more carefully.

The studies found that algorithm animation was less effective in novice students who were learning the algorithms for the first time, yet could be an effective tool for students who are looking to revise particular concepts. Consequently, any future algorithm animation tool should likely act as a secondary learning resource. Ideally, students should have been taught the concepts using conventional methods to begin with, then the educator can distribute the application to students who can utilise this as a means to refine and clarify understanding.

When considering the previous guidelines proposed by Stasko, Badre and Lewis and how they might affect algorithm animation software, it appears a potential system should:

- Provide supplementary textual explanations that rationalise and justify the logic of the algorithm.
- Include rewind-replay functionality that allows the animation to be restarted, or played step-wise, backwards and forwards.
- Ensure users can create the animation themselves by allowing them to modify the input to the algorithm.

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.

However, a tool based on compiler animation would need to include both aspects of complexity and usability. The tool would implement an algorithm that is certainly more complex and volatile than even Dijkstra's

algorithm, but in order to be educationally effective, also needs to provide the display of highly input-dependent analytics and implement the usability and interactivity features such as playback control that are embedded within the guidelines of the previous studies.

Chapter 3

The Fun Programming Language & Compilation Theory

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

3.1 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 [1], 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.

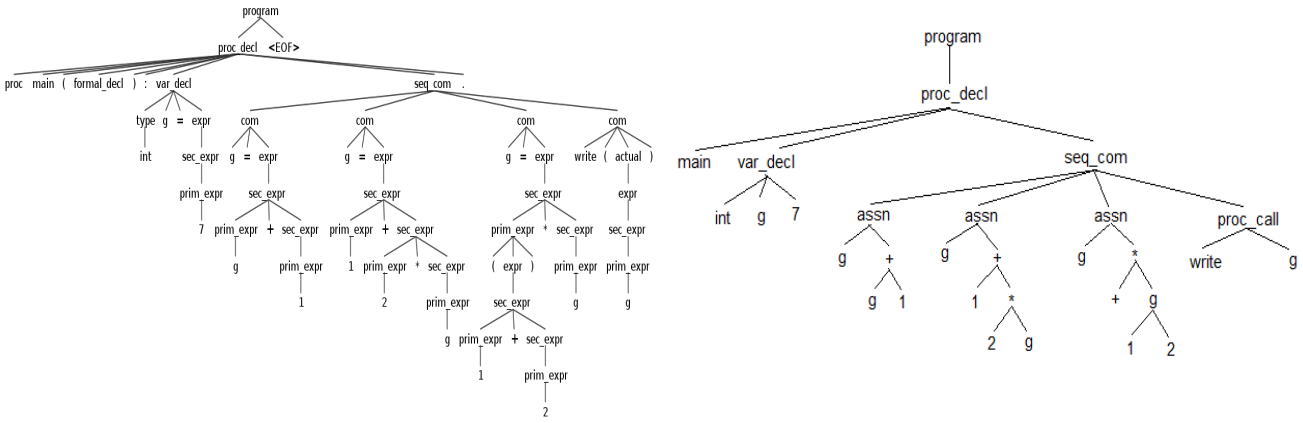
Whilst Fun may differ significantly to other programming languages, particularly in its complexity; it is not the case that the concepts illustrated are exclusive to the Fun language or the Fun compiler. Fun is sufficiently generic that the core concepts can be explained in an easily understandable format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

3.2 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 a 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 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

3.3 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.

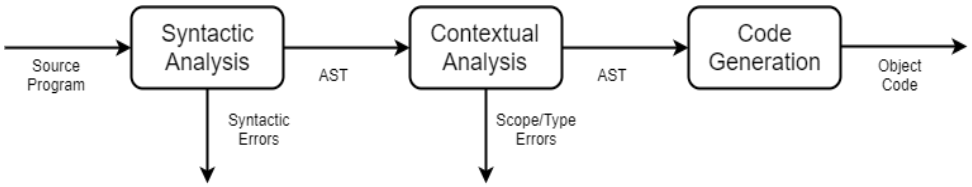


Figure 3.2: Compilation Pipeline

3.3.1 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.

3.3.2 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 into *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.

3.3.3 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.

Chapter 4

Requirements

This chapter initially discusses the methods via which the requirements of this project, which we will henceforth refer to as the *FunCompiler*, were collected and established. The remainder of the chapter lists the FunCompiler's user stories, functional requirements and non-functional requirements.

4.1 Methodology

The requirements of the FunCompiler were elicited through two main techniques. Firstly, through multiple interviews with Simon Gay. Simon is the current lecturer of the Programming Languages course at Glasgow University. Since both the Fun language and the compilation of Fun programs are delivered during the Programming Languages course, students enrolled in this course will naturally be the target user-base of the FunCompiler. Thus, Simon clearly has the expertise and the experience to offer valuable recommendations on the function and operation of the FunCompiler.

The second technique was to simply utilise the insight gained from performing background research. In particular, we strive to ensure the requirements meet the three guidelines we saw in Section 2.1, proposed by Stasko, Badre and Lewis. Below, the three guidelines have been rephrased to apply directly to the FunCompiler:

Guideline a) Provide supplementary textual explanations that rationalise and justify the logic of the compiler as it visits each node in the AST.

Guideline b) Include rewind-replay functionality that allows the compilation animation to be restarted, or played stepwise, backwards and forwards.

Guideline c) Ensure users can create the animation themselves by allowing any arbitrary Fun program as input to the compiler.

4.2 User Stories

After conducting the interviews and the background research, a set of user stories was devised. User stories are short and simple descriptions of a feature, told from the perspective of a potential user:

User Story	Description
1	As a user, I want to read details of the Fun language, so that I can write valid Fun programs as input and better understand the compilation animations.
2	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.
3	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.
4	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.
5	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.
6	As a user, I want to be able to see textual descriptions that explain what the compiler is doing at each stage of the animation, so that I can better understand the logic of the compiler.
7	As a user, I want to be able to see any supplementary information or feedback (including address/type tables, code templates and generated object code), so that I get all the information available in order to help further my understanding.
8	As a user, I want to be able to replay an animation, so that I can review any details I missed/misunderstood.
9	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.
10	As a user, I want the option to select a pre-defined Fun program from a list of examples, so that I can better understand the possible types of Fun program.

4.3 Functional Requirements

After creating user stories and using any previous research, a formal list of functional requirements was created. Functional requirements are intended to capture a specific function of a system:

Along with implementing the core functionality of a compiler animator (along with a few extra features), it

Functional Requirement	Description
1	Allow users to input any arbitrary Fun program as input to the compiler animator.
2	Allow users to animate either contextual analysis or code generation separately.
3	Display the generated AST that represents the input Fun program.
4	Enable a controllable animation over this AST representing one of the two phases (contextual analysis or code generation).
5	At each step of the animation, highlight the corresponding node in the AST.
6	Allow users to play the animation continuously.
7	Allow users to pause the animation.
8	Allow users to move forwards through the animation, one step at a time.
9	Allow users to move backwards through the animation, one step at a time.
10	At each step of the animation, display messages that explain the logic of the compiler. I.e., what it is currently doing, or what it is going to do.
11	At each step of the animation, display the type table (if contextual analysis) or the address table (if code generation) in its current state during the compilation.
12	During code generation, display the code template of each node as it is visited.
13	During code generation, display the emitted object code in its current state during the compilation.
14	If a user inputs a syntactically invalid Fun program, prevent the animation and report the errors to the user.
15	If a user inputs a contextually invalid Fun program, allow animation of the contextual analysis phase but disallow animation of the code generation phase and report the errors to the user.
16	Make the full specification of the Fun language available within the web application.
17	Allow users to select an example Fun program from a list, which will be automatically inserted into the input area.

is clear to see from functional requirements **1**, **6**, **7**, **8**, **9** and **10**, that we have more than comfortably included the elements necessary to satisfy guidelines a), b) and c).

4.4 Non-functional Requirements

In contrast to functional requirements that detail specific behaviours or a system, non-functional requirements often consider overall utilities of a system, such as security, usability and extensibility:

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

4.5 Analysis

In overview, a typical workflow for the FunCompiler might look as follows. Firstly, the user arrives at the website and should be able to view the specification of Fun language before needing to use the animator. Then, the user can type a Fun program into some kind of code editor and choose to animate either contextual analysis or code generation. At this point, if there were any syntactic or contextual errors, they would be reported to the user using the semantics as defined above.

After selecting one of the two phases, the AST that represents the input program is displayed on screen along with playback buttons (play, pause, forwards, backwards). As the animation progresses, node within the AST are highlighted to symbolise the current progress of the compiler. As each node is highlighted, detailed information as described above is simultaneously displayed elsewhere on screen. The user is of course free to pause the animation to review the current information displayed, or rewind to review. At any point the user may modify the current input Fun program and select either the same or a different compilation phase, which will halt any current animations.

This potential workflow of the FunCompiler is enough to meet all functional requirements. The non-functional requirements cover issues that are strictly more technical, the solutions to which will be discussed during the system design and the physical implementation.

Chapter 5

Design

This chapter covers the aesthetic design process of the FunCompiler. We start by discussing the main design goals we wish to achieve. Then, we illustrate and describe the individual components of the interface. Following this, we give a more complete design and use a "storyboard" to demonstrate the workflow of the website. Finally, we discuss how we might design the application to be responsive on smaller screens.

5.1 Design Strategy

To recap, the FunCompiler allows the input of a Fun program, then generates an animation (equipped with playback controls) and finally augments this animation with supplementary analytical information pertaining to the compilation being animated. We can group these aspects into three sections: the input, the animation and the analytics.

Given the educational nature of the FunCompiler, it is important that students are able to cross-reference between these three sections in order to evaluate the relationships between them. For example, when an AST is displayed during an animation, the student clearly needs to retain access to the corresponding input program in order to reason about the structure of the AST.

Due to this inherent link between the input, the animation, and the analytics; it was determined that the core design philosophy of the FunCompiler's user interface would be to ensure that each of these components is displayed simultaneously within the same view (web-page).

5.2 Interface Components

This section

Given everything discussed so far, we can attempt to create a list of the individual components that will likely be required within the user interface of the final application:

- A section to display the fun specification.
- A code editor.
- Buttons to select contextual analysis or code generation.

- A section to display the animation.
- A set of playback controls.
- A section to display explanatory messages during the animation.
- A section to display type/address tables.
- A section to display code templates (code generation only).
- A section to display object code (code generation only).
- A navigation bar.

5.2.1 The Fun Specification

Figure 5.1: Wireframe

Appendices

Bibliography

[1] ANTLR. Antlr. `http://www.antlr.org/`. [Online; accessed 10-February-2018].

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

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

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

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

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

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