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 typically unnecessary in a modern environment, being required only in a small number of highly specialised industries. Nevertheless, remaining as a cornerstone of a computer science curriculum at many schools and universities is the art of compiler construction, behaviour and optimisation. Why is this?

When questioned on the necessity of teaching compilation, 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*" [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. To gain a true understanding of the building blocks of computer science is to gain a greater insight into the scientific and mathematical concepts within the field, which provides the most effective means of making informed technological decisions.

Compilation theory is a foundational component 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.

1.1   Motivation

Compilation can often be a challenging field to teach effectively. Many elements of compilation involve the generation and traversal of complex, abstract data structures. Abstract, conceptual notions such as these are notoriously difficult both for students to understand and for educators to explain. We desire to find a solution to this problem by creating techniques that provide concrete representations of these types of abstract ideas.

As a common approach to this issue, an educator will attempt to visualise the data structures and the traversals over them. The educator usually does this by creating some sort of "slideshow", using an application such as Microsoft PowerPoint. The slides contain a pictorial representation of the data structure in question, and each slide demonstrates a distinct step of the traversal over that data structure. For example, an educator seeking to teach students about a depth-first traversal over a binary tree might design their slideshow similarly to Figure 1.1, where we see that each consecutive slide illustrates the next step of the traversal by highlighting the corresponding node.
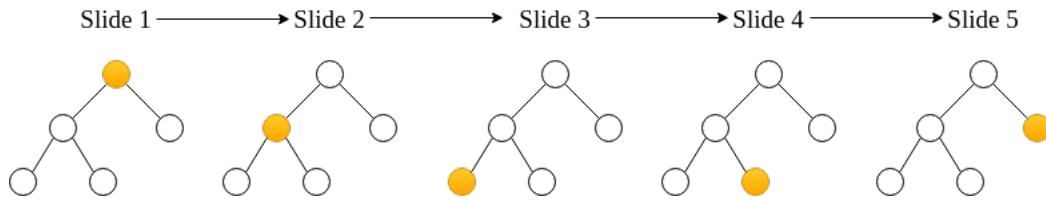
Figure 1.1: A sequence of slides demonstrating a depth-first traversal over a binary tree

Whilst using a slideshow to demonstrate these concepts does provide part of the solution to representing abstract notions, they do have several limitations. Firstly, to create visualisations in this way is an arduous task on behalf of the educator, realistically meaning that the length of the demonstration must remain short. Secondly, there is a limited amount of control over the playback of the slides in regards to playing, pausing, rewinding and restarting; which of these functions is possible depends on the choice and availability of slideshow application. Thirdly, and most importantly, the educator is restricted to showing pre-determined examples only. In the case of compilation, not only can the input be completely arbitrary, but the semantics of the compiler varies for each unique construct in the source language. This means 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).

Contextual analysis and code generation are two different stages of compilation which aim to validate certain aspects of the compiler's input and construct some appropriate output. The behaviour of these stages is expressed by traversals over an AST. An AST is a tree-based data structure that represents the hierarchical syntactic structure of a segment of code. Fun is a simple, educational programming language equipped with a compiler. Chapter 3 introduces each of these concepts in considerably more detail.

The application will allow a user to supply a program written in the Fun language as input to the Fun compiler. The user can then choose to animate either the contextual analysis or code generation phase of the resulting compilation. In both cases, a visualisation of the input program, represented as an AST, will be presented to users. In order to demonstrate how the compiler traverses the AST during these phases, nodes within the tree will be highlighted consecutively corresponding to the progression of the traversal, akin to the method displayed in Figure 1.1. In general, the application should:

- Allow a user to input any program written in the Fun language.

- Animate the contextual analysis phase of a program's compilation.

- Animate the code generation phase of a program's compilation.

- Augment the animation with explanatory messages that provide insight into the internal logic of the compiler.

- Augment the animation with any other relevant supplementary information.

- Allow the animation to be played, paused or stepped through (backwards and forwards).

2

The application, which will act as an educational tool, resolves many of the problems we previously discussed with respect to slideshow-based visualisations. Most obviously, educators are no longer required to create the visualisations themselves, a time consuming process. They can simply project footage of the application to students within a classroom setting, or even better, direct students to access the application themselves. The application aims to implement playing, pausing, rewinding and restarting of the animation, meaning the previous issues of playback limitations are removed as long as the user has access to a web browser. The application also allows any arbitrary Fun program to be animated, meaning the restriction of only showing examples with a pre-determined input is eliminated. Additionally, the application desires to supply in-depth automated analytical details (in the form of explanatory messages and supplementary information), something that is not currently possible by present techniques. The application 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 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 paragraph 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 two separate groups of students the concept of a "pairing heap", with one group using textual descriptions of the algorithm and the other using an animation of the algorithm. Stasko then repeated a similar
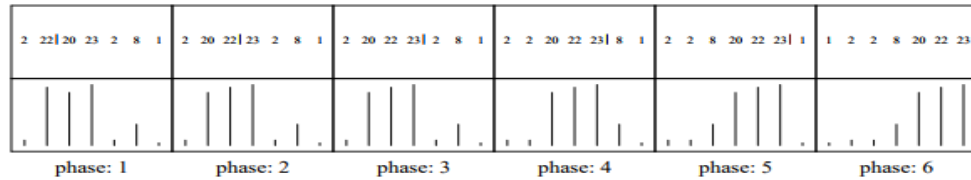
Figure 2.1: A sequence of stills from an insertion sort algorithm

experiment in 1999 with Byrne and Catrambone. Again, one group used an animation but this time the other group used static diagrams (such as figures 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 shown similar results. The general consensus being that whilst animations do provide a small educational 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 particularly effective when students are trying to learn a concept for the first time, but are 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 algorithm's current progress in both plain English and pseudo-code.
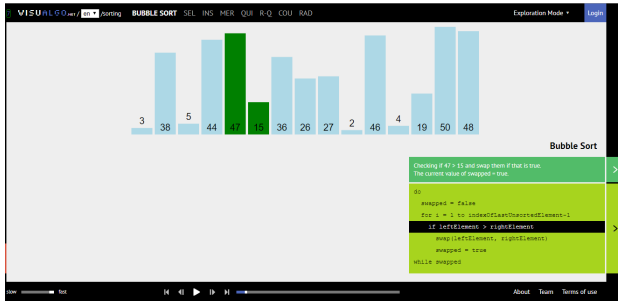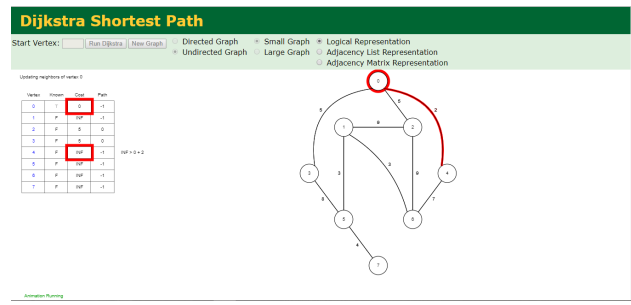
(a) VisuAlgo   (b) The University of San Francisco

Figure 2.2: Animations of different algorithms from different applications

Departing 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 most 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 cannot define your own input and you cannot 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 be 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 when applied to 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, it 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 introduce the reader to the main tools and concepts used throughout this dissertation. Firstly, we begin with a small overview of the Fun programming language and its main features. Secondly, we consider how and why compilers parse source code into syntax trees and why some representations of a syntax tree might be more beneficial than others. Thirdly, we take an in-depth look at the three main phases of compilation: syntactic analysis, contextual analysis and code generation; the latter two of which form the animation basis of this project. Finally, we discuss the use of code templates as a theoretical tool for assisting compiler developers in deciding which object code to generate for various constructs within a programming language.

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" which 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 Java and ANTLR [1], developed at Glasgow University by David Watt and later extended by Simon Gay. Its purpose is to exhibit 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.

Fun has variables, procedures and functions. Variables can have a declared type of `int` or `bool` only. All variables must be initialised with an expression of the same type upon declaration. Procedures have type `T` $\rightarrow$ `void` and functions have type `T` $\rightarrow$ `T'`. `T` represents the type of an optional parameter. Procedures return no value (hence $\rightarrow$ `void`) and functions must return a value of the type `T'` (hence $\rightarrow$ `T'`). Fun has one predefined function and one predefined procedure: `read` and `write`. `read()` is a function that returns user input as an integer and `write(`**`int`**` n)` is a procedure that writes the value of `n` to standard output. Below is an example of a Fun program which utilises variables, functions and procedures to calculate and output the factorial value of user input:

```
bool verbose = true

func int fact(int n):
    int f = 1
```

```
    while n > 1:
        f = f*n
        n = n-1   .
    return f
.


proc main():
    int num = read()
    while not (num == 0):
        if verbose: write(num)  .
        write(fac(num))
        num = read()  .
.
```

Similarly to other languages, the entry-point of a Fun program is the `main()` procedure. In this example, `main()` begins by reading a number from user input. If the number is 0 the program terminates, otherwise, if the boolean variable `verbose` is set to `true`, `main()` writes the number to standard output and then passes the number to the `fact()` function. `fact()` iteratively calculates and returns the factorial value of its input argument `n`. `main()` then writes the factorial value to standard output and requests another number from the user.

The Fun programming language has a ***flat block structure***. A flat block structure means that variables may reside in either a local or a global scope, and that the same identifier may be declared locally and globally. Since functions and procedures cannot be defined within other functions and procedures, they are always global. A variable that is declared as a parameter of or declared within a function/procedure is considered local. Variables declared outside of any function/procedure are considered global. In the example above, we note that `verbose` is a global boolean variable, whereas `n` and `f` are integer variables local to the function `fact`.

Whilst Fun may differ significantly from other programming languages, particularly in its complexity, it is not the case that the notions it aims to represent are exclusive to the Fun language or the Fun compiler. Fun is sufficiently generic that the core concepts of compiler theory can be delivered in a simple, comprehensible 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

During compilation, a source program is parsed into a ***syntax tree***. A syntax tree is a hierarchical syntactic representation of a source program; with top-level statements nearer the root, and more deeply nested statements nearer the leaves. We typically consider two types of syntax tree: the ***parse tree*** (sometimes called a concrete syntax tree) and the ***abstract syntax tree*** (AST). Translating a source program into a syntax tree means the compiler can more easily reason about the underlying structure and grammatical nature of the program, which is necessary for certain stages of compilation.

A parse tree retains all information about a program, including the information that may appear to be unnecessary, such as white-space and parentheses. Conversely, an AST is a smaller, more concise adaptation of the parse tree. An AST usually ignores redundant details which are derivable from the shape of the tree. Figure 3.1 demonstrates the visual differences between a parse tree and an AST of the same hypothetical Fun program.

At the initial stages of compilation, the compiler will attempt to translate the input program into either a
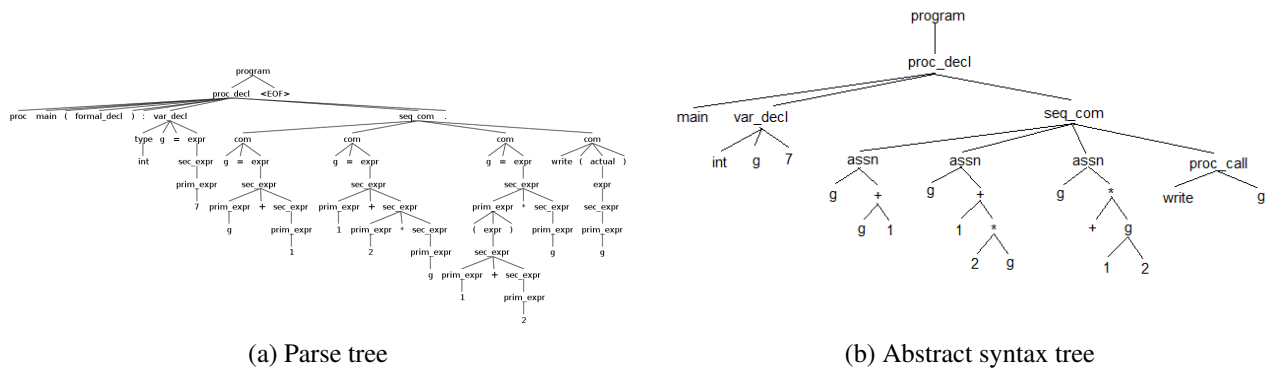
9

(a) Parse tree   (b) Abstract syntax tree

Figure 3.1: Parse tree and AST of the same hypothetical Fun program

parse tree or an AST. The generated tree is then traversed during contextual analysis and code generation, which is described in detail in Section 3.3. How the compiler chooses to visit each node in the tree during the traversal is language-dependent, but the semantics are vaguely similar to that of a depth-first traversal.

Since ASTs are usually much smaller and less crowded than their parse tree counterparts, they are often far easier to read and understand. Any information that is lost from the conversion of a parse tree to an AST is usually semantic and does not affect how the syntax tree would be evaluated. Consequently, when trying to illustrate syntax tree concepts to students from an educational perspective, we generally prefer to use visualisations of ASTs over parse trees.

3.3   Compilation Phases

In general, 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 or contextual analysis encounters an error (as determined by the language's specification) during its execution, the phase completes, but the remainder of the compilation process is halted and the errors are reported to the programmer. Note that the following subsections assume the compiler is employing an AST representation of a syntax tree. See Figure 3.2 for a diagram of a typical compilation pipeline.
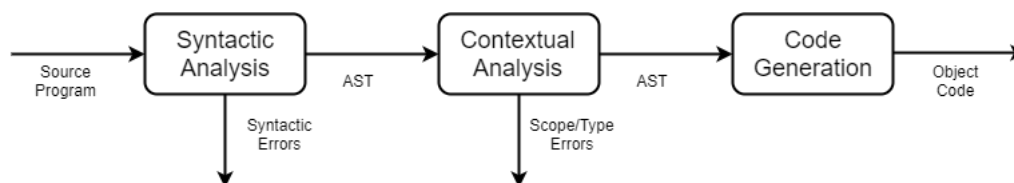

Figure 3.2: Compilation Pipeline

10

3.3.1 Syntactic Analysis

Syntactic analysis takes a source program as input and upon success, produces an AST. The purpose of syntactic analysis is to verify whether the source program is well-formed in accordance to the source language's syntax rules. 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*. A token is simply a single element of the input program. For example, a token could be an individual identifier, operator or keyword. How the compiler chooses to define tokens is dependent upon the implementation of the language. The token stream is then passed as input to the parser. Figure 3.3 shows how a small excerpt of code may be decomposed into a token stream.
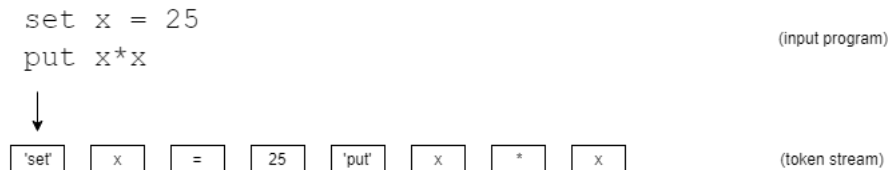

Figure 3.3: Lexing of source code into a token stream

A parser converts a token stream into an AST using some parsing algorithm. The Fun compiler uses *recursive-descent* parsing. Recursive-descent parsing involves "consuming" the token stream from left to right. At each token, the parser checks whether the next sequence of tokens are of the correct type, as determined by the language's syntax. The parser carries out this check for every token in the stream. If any checks fail, a syntax error is reported to the programmer. If all tokens are consumed successfully, the parser outputs an AST representing the parsed program.

3.3.2 Contextual Analysis

Upon successful completion of syntactic analysis, the generated AST is traversed by a contextual analyser. A contextual analyser checks whether the source program represented by the AST conforms to the source language's scope and type rules. Contextual analysis utilises an auxiliary data structure called a *type table* in order to help carry out these checks. Each row of the type table contains three fields of information about a declared variable: its scope (which as discussed in Section 3.1 can be local or global), its identifier and its type. Figure 3.4 demonstrates a small example. The table reveals that the corresponding program contains two integer variables with the identifier x (globally and locally defined) and a global procedure with the identifier main which takes no parameters and returns no value. Contextual analysis can be broken down into *scope checking* and *type checking*.

Scope	Identifier	Type
global	'x'	INT
global	'main'	VOID -> VOID
local	'x'	INT

Figure 3.4: Type Table

Scope checking ensures that every variable used in the program has been previously declared. If the contextual analyser encounters the *declaration* of an identifier as it is traversing over the AST, it inserts the identifier along with its scope and type into the type table. If the contextual analyser finds that there is already an entry in the table with the same scope and identifier, then a scope error is reported to the programmer. Similarly, if

the contextual analyser encounters the *usage* of an identifier during the traversal, it checks that the identifier is already in the type table (i.e., has been previously declared). If the identifier cannot be found in the type table, then a scope error is reported to the programmer. Figure 3.5 shows two ASTs that illustrate the declaration and usage (assignment) of a variable n. The variable *declaration* construct in Figure 3.5a would lead the compiler to insert a row into the type table with values: `[global, n, INT]`. The variable *assignment* construct in Figure 3.5b would lead the compiler to lookup entries in the type table with identifier n to ensure the variable has been previously declared before attempting to assign a new value to it.

(a) Variable declaration   (b) Variable usage

Figure 3.5: ASTs demonstrating the declaration and usage of a variable n (assuming global scope)

Type checking ensures that every operation in the program has operands of the expected type. The rules the contextual analyser uses to determine if an operation has the "correct" operands vary from construct to construct. For example, referring back to Figure 3.5b, we see the construct for variable assignment. In this case, we are assigning the value of some expression, represented by `expr`, to the variable n. When the contextual analyser encounters this construct during the traversal, it will retrieve the type of n from the type table, then traverse the expression `expr` to determine its type. The contextual analyser will then check that the type of `expr` is the same as the type of n. For example, given that in Figure 3.5a n was defined as an integer, `x = 5 + 10` would be a valid assignment, whereas `x = false` would result in a type error.

3.3.3 Code Generation

Upon successful completion of contextual analysis, the generated AST is traversed by a code generator. A code generator translates the source program into a lower level language, such as assembly language or object code. Code generation utilises an auxiliary data structure called an ***address table***. Each entry in the address table contains three fields of information about a declared variable: its scope, its identifier and its address. Figure 3.6 demonstrates a small example. The table reveals how each identifier has been allocated an address in the address space belonging to either local or global variables. Code generation can be broken down into ***address allocation*** and ***code selection***.

Scope	Identifier	Address
global	'x'	0
global	'main'	7
local	'x'	2

Figure 3.6: Address Table

Address allocation decides the representation and address of each variable in the source program. If the code generator encounters the declaration of an identifier as it is traversing over the AST, it determines a suitable address which it inserts into the address table along with the corresponding identifier and scope. In the case of Fun, all variables conveniently have a size of 1, meaning the compiler can simply allocate incrementally consecutive addresses to each variable as it is encountered.

Code selection selects and generates the final object code. The developer of the compiler should plan which

object code is selected for each construct encountered within the AST. The developer does this by devising a *code template* for each construct in the language.

3.4   Code Templates

When developing a compiler, it is important to consider how each construct in the underlying language should be translated into object code. A code template provides a means of creating a theoretical model for each construct that specifies how the corresponding object code should be selected. Each template is often just a list of directives (written in plain English) that specifies which object code instructions to emit and defines how any constituent expressions/commands of the construct should themselves be considered for code selection.

For example, on the left-side of Figure 3.7 we see an AST that demonstrates the addition of two expressions, where `expr1` and `expr2` are sub-expressions of the `PLUS` construct. Of course, the value of the two expressions must be calculated before attempting to add the two together. In other words, the compiler must evaluate `expr1` and `expr2` and emit the object code to load those values into memory *before* emitting the addition object code. Therefore, the code template for the `PLUS` construct should state that `expr1` and `expr2` are to be evaluated before we emit any object code to add the expressions together.

On the right-side of Figure 3.7 we see an illustration of the code template as described above. Considering each instruction sequentially, it stipulates that firstly we should generate the object code for `expr1`, then `expr2`. Only after emitting both instructions that load the value of these expressions into memory can we emit the `ADD` object code instruction.
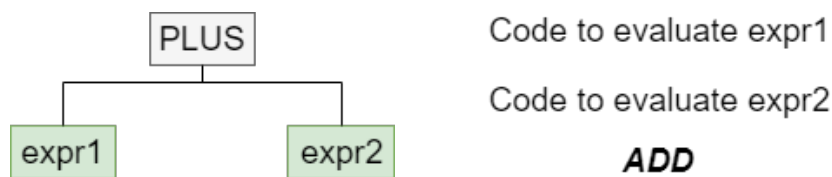

Figure 3.7: AST and code template of a `PLUS` construct

In the general case, it is often true that the compiler may need to translate some (or all) of the constituent statements of a construct into object code before emitting any object code specific to the original construct itself. Unfortunately, how each construct should be evaluated in regards to code generation is not always immediately clear just from the structure of the AST. This is the scenario in which code templates are practical, as they provide a way to exemplify how each construct in the source language should be translated into object code.

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.

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 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).

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.

4.4   Non-functional Requirements

In contrast to functional requirements that detail specific behaviours of 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, nodes 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 develop simple low-fidelity wireframe designs of the individual components of the user interface and consider how each of these different components might satisfy our functional requirements. We then look at a more complete design of the user interface and observe how a user might transition from page to page.

5.1 Application Interface

To act as the first step in realising the project's functional requirements, we develop a series of quick and primitive low-fidelity wireframe designs, each representing an individual component of the user interface. Detailed below are a set of five interface components which when combined satisfy the majority of the application's functional requirements. Each component consists of a wireframe design and considers how the component may meet one or more of the functional requirements (FRs).

5.1.1 Code Editor

The code editor will be the text-area in which users are able to enter code written in the Fun language (FR **1**). Ideally, the code editor should follow some of the semantics of standard programmatic text editors, including syntax highlighting and auto-indention. Notice that in Figure 5.1 two buttons for "Contextual Analysis" and "Code Generation" have been attached to the bottom of the code editor. These buttons will trigger the transmission of the input code to the compiler and thus initiate the corresponding animation (FR **2**).
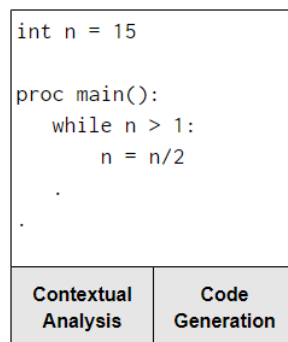

Figure 5.1: Code editor wireframe

5.1.2 AST

The AST of the input program is the foundation of the animation. The AST will be displayed in the conventional hierarchical tree format where the animator will highlight each node consecutively, corresponding to the progress of the compiler (FR **3, 5**). For example, along with showing the general structure of the AST for a Fun program, Figures 5.2a to 5.2c demonstrate how the application might animate the AST. At each "step" of the animation, the next node in the traversal is highlighted, whilst the previous node is returned to its original state.

(a) Step 1 of animation   (b) Step 2 of animation   (c) Step 3 of animation
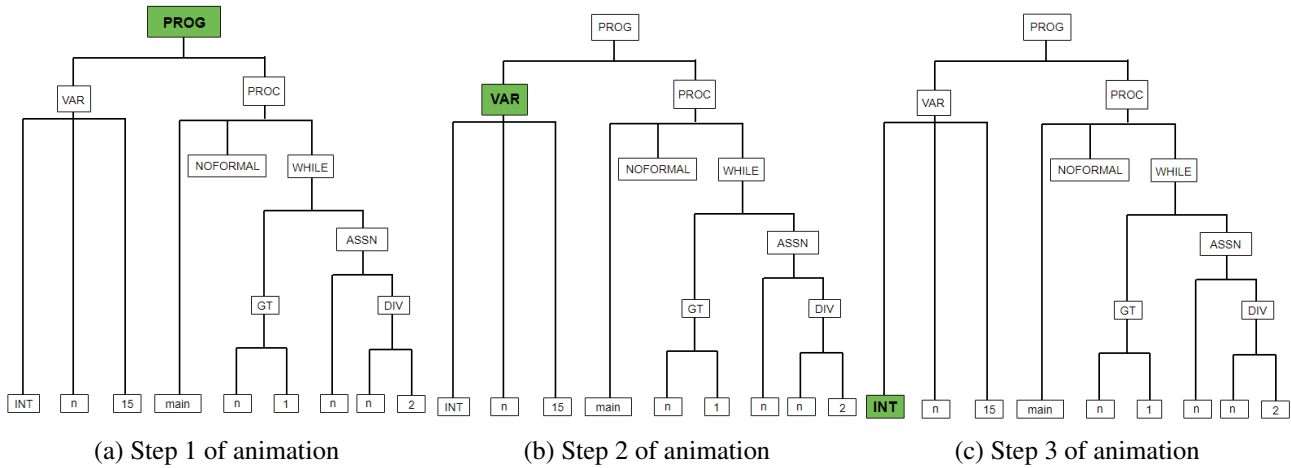
Figure 5.2: AST wireframes

5.1.3 Augmentations

As the compiler traverses the tree, the application should augment the animation with any supplementary information that may aid the user's understanding of the compilation, including explanatory messages, type/address tables, code templates and object code. We henceforth collectively refer to each of these factors as "augmentations". Figure 5.3a illustrates the augmentations required for contextual analysis, including a type table and a section to display explanatory messages (FR **10, 11**). Figure 5.3b illustrates similar augmentations for code generation, with the type table replaced by an address table and two added sections to display code templates and object code (FR **12, 13**). Note that all augmentations, other than tables, are just textual descriptions/lists, and hence are simply represented as labelled areas in the following wireframes.
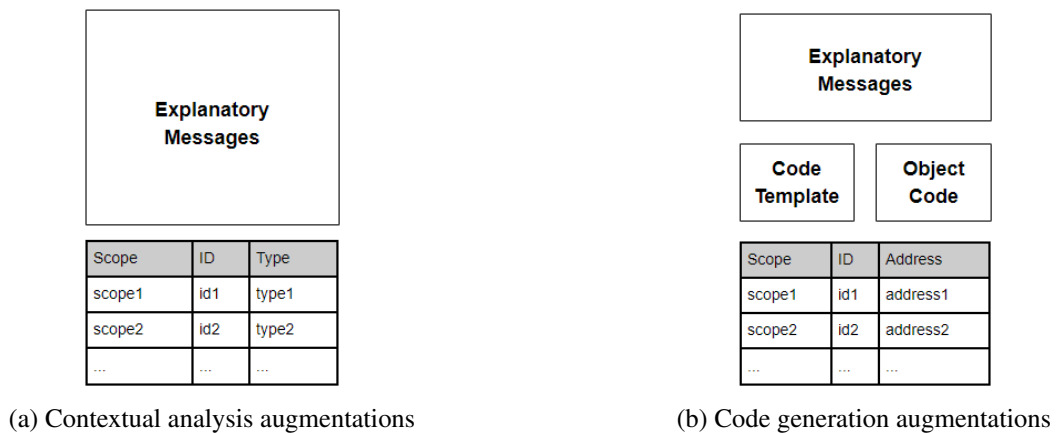

(a) Contextual analysis augmentations   (b) Code generation augmentations

Figure 5.3: Augmentations wireframe

19

5.1.4 Playback Controls

Users require the ability to play, pause and step through (backwards and forwards) the animation. Figure 5.4 shows two sets of playback buttons (FR **4**). Intuitively, when the play button is pressed, it should be replaced with the pause button and vice versa. If the user presses the play button, the animator should highlight the next node in the sequence at regular time intervals (for example, each second) (FR **6**). If the user presses the pause button, the animation should be halted and the currently highlighted node should remain highlighted (FR **7**). If the user presses the forwards/backwards buttons, this will highlight the next/previous node in the sequence, one node per button press (FR **8, 9**). If the animation is currently playing when a forwards or backwards button is pressed, it should be implicitly paused.

Figure 5.4: Playback controls wireframe

5.1.5 Fun Specification

Figure 5.5, illustrates the Fun specification broken down into seven sections. Tabs along the top of the component allow the user to navigate between each section. Each section contains varying amounts of information to explain each concept (FR **16**).
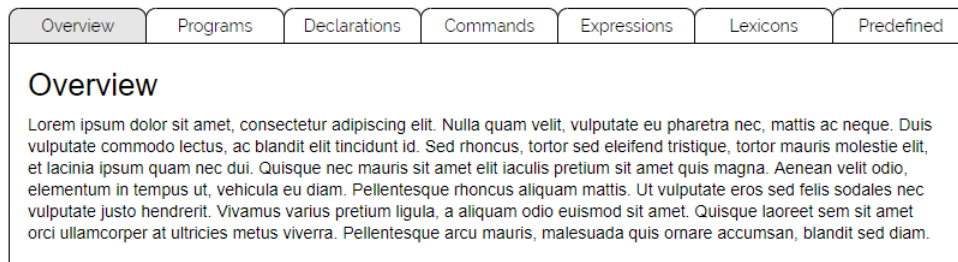

Figure 5.5: Fun specification wireframe

5.2 Full Design

With all necessary components designed, we consider how to combine each of these to create a set of full-sized designs. The presiding design philosophy is that the code editor, the animation and the augmentations are all available within the same view. This is important for two reasons. Firstly, from an educational perspective, there is the obvious necessity that users are able to easily cross-reference between these three components. Secondly, from a usability aspect, as an animation is running, a user should be able to modify the input code and generate a new animation without having the navigate to a different page.

We partition the FunCompiler by splitting the application into three separate pages: the landing page (containing the Fun specification), the contextual analysis animation page and the code generation animation page. We also insert a small navigation bar on each page which includes a link to return to the landing page.

5.2.1 Landing Page

Figure 5.6 demonstrates the landing page of the FunCompiler. Here we see that the user has access to the code editor along with access to the full specification of the Fun language. A user can enter a Fun program of their own creation then choose to animate either contextual analysis or code generation by clicking one of the buttons below the code editor.
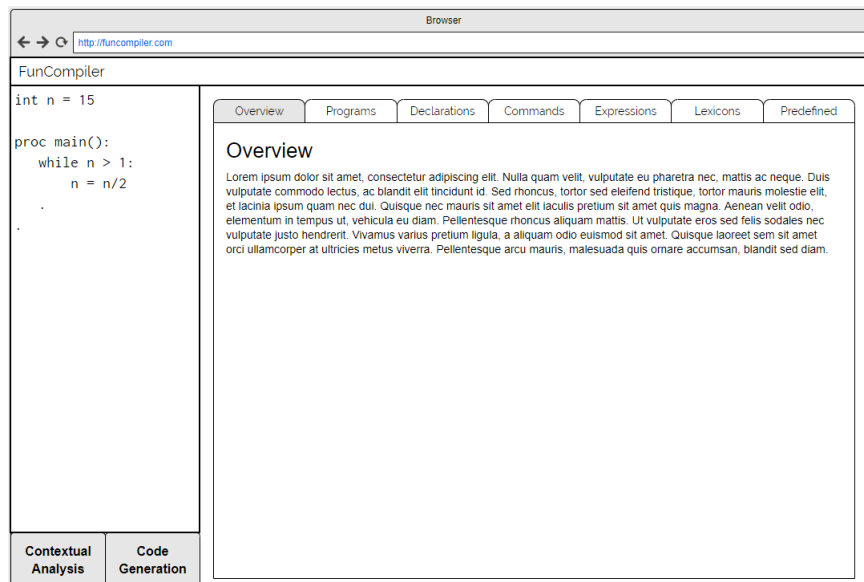

Figure 5.6: Landing page wireframe

5.2.2 Contextual Analysis Animation Page

If the user chooses to animate contextual analysis, they are redirected to the page illustrated in Figure 5.7. This page displays an AST generated from the input program, a set of playback controls for the animation, an area to output explanatory messages and a type table. From this page the user still has access to the code editor and is free to modify the input program and generate a new animation.
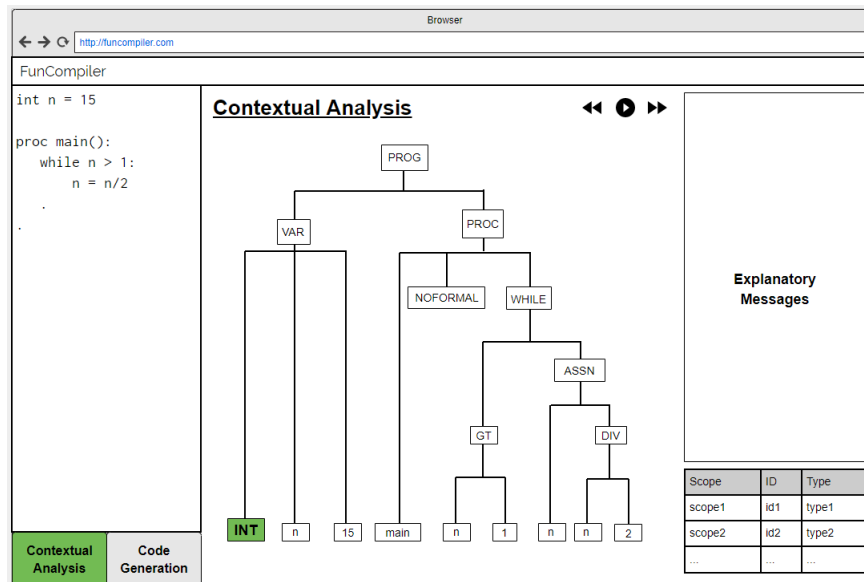
Figure 5.7: Contextual analysis page wireframe

5.2.3 Code Generation Animation Page

If the user chooses to animate code generation, they are redirected to the page illustrated in Figure 5.8. This page displays very similar details to the contextual analysis page, except the type table is swapped for an address table and sections to display code templates and object code have been inserted.
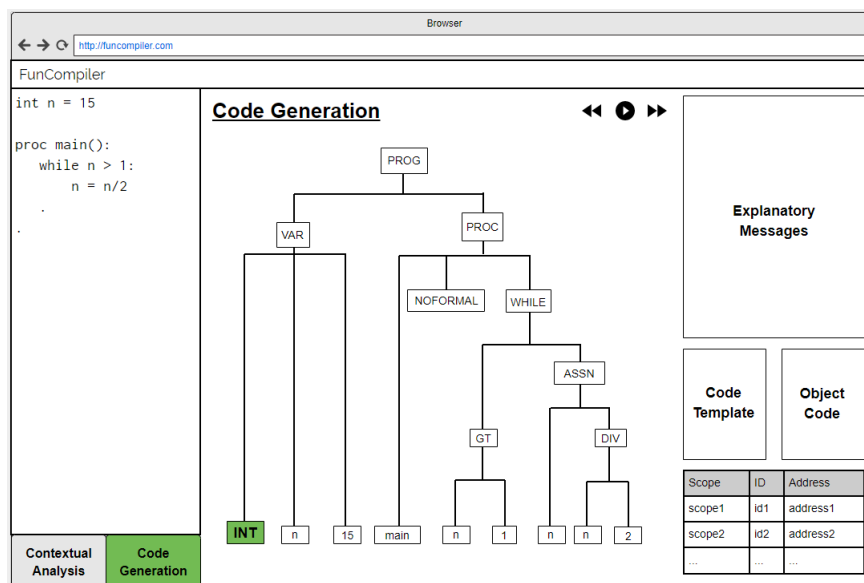

Figure 5.8: Code generation page wireframe

5.3 Error Handling

The above component designs successfully covered all but two of the 16 functional requirements. The remaining functional requirements (FR **14, 15**) are concerned with error handling and preventing certain actions in the event of errors. More specifically, FR **14** states that we must not allow the animation of either contextual analysis or

code generation in the event that the input Fun program contains syntax errors. FR **15** states that if the input Fun program contains contextual errors, we may animate contextual analysis, but not code generation. In both cases, we present the user with the relevant errors if they try to execute a prohibited action. In order to retain space for the more important aspects of the interface, when a user attempts to submit an invalid program, we simply display the errors in a pop-up window without disturbing the core contents of the page, as illustrated in Figure 5.9. With this final addition in place, we have successfully realised all of the FunCompiler's functional requirements.
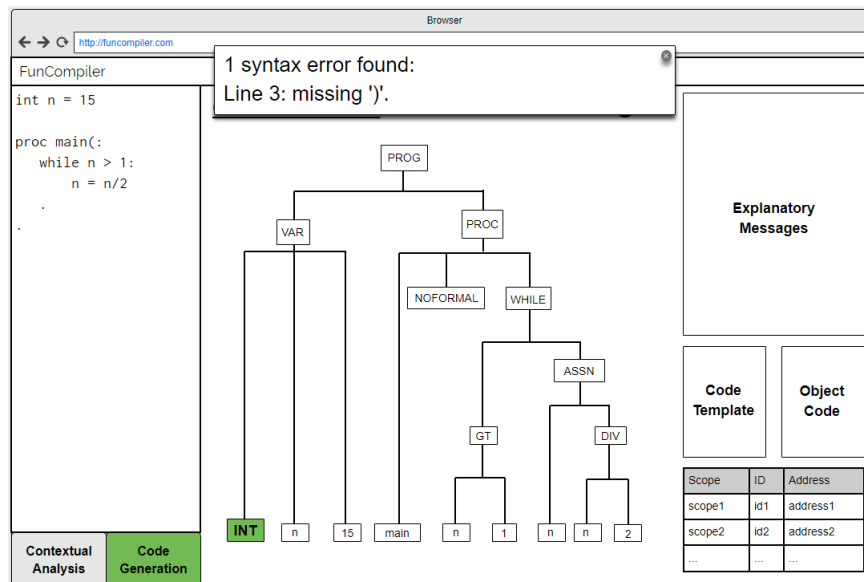

Figure 5.9: Syntax error wireframe

Chapter 6

System Architecture

In this chapter we define the system architecture of the FunCompiler. That is, we establish individual system-level components and build a model to specify their organisation/structure, alongside showing the patterns of communication between them.

Firstly, we discuss the differences between a monolithic and a micro-service architecture in regards to web-applications. Secondly, we consider which of these architectures is most applicable to the FunCompiler and how the components of the system will be arranged to fit that model. Finally, using a bottom-up approach we describe the specific technologies that will be used within the FunCompiler, from application-level development frameworks to build tools and packaging systems.

6.1 Monolithic vs. Micro-service Architecture

Traditionally, websites have been built as large, monolithic applications, where each unique service within the website (e.g., authentication, routing, API exposure) is encapsulated under one application, as illustrated in Figure 6.1a. Whilst a monolithic architecture may be an appropriate choice for many basic applications, if the system has a diverse range of service implementations, issues are likely to occur.

For example, consider the scenario of a monolithic online shop. Suppose the vast majority of the website's business logic is written in Python but the authentication service is written in PHP. To force the PHP service to operate within a Python environment not only clearly violates the basic principles of a *separation of concerns*, but likely leads to inefficient/insecure methods of communication (e.g., writing/reading from files) and an unscalable, tightly coupled code-base. To solve this issue whilst maintaining a monolithic architecture would require translating the authentication service from PHP to Python, however, this may be undesirable or simply infeasible.

The emerging trend is to extract each of these services out of the main application and allow them to behave as stand-alone components, referred to as micro-services. Figure 6.1b demonstrates how a user interacts with the web-application, which in turn communicates with the individual micro-services that have been separated into their own components. Employing a micro-service architecture decouples the services, handles scalability issues with greater ease and allows diversity in programming language or web-framework selection.
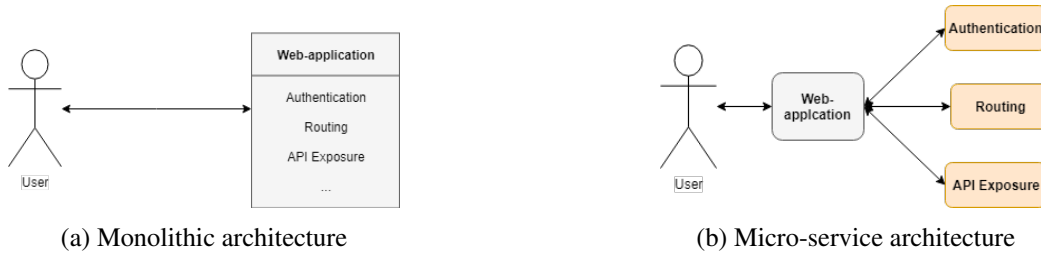
(a) Monolithic architecture　　　　　　　(b) Micro-service architecture

Figure 6.1: Monolithic vs. micro-service architecture

6.2 FunCompiler Architecture & Communication

As discussed in Chapter 3, the compiler of the Fun language is a pre-existing piece of software built with Java and ANTLR. In order to incorporate the compiler within a web-application we have three choices:

a) Translate the compiler into the same language as used to implement the web-application.

b) Implement the web-application using a Java-based web-framework.

c) Employ a micro-service architecture, partitioning the FunCompiler into two distinct units: the web-application and the compiler.

Although all choices do essentially produce the same result, choice a) leads to a significant time investment that may produce a weaker implementation of the compiler and choice b) compels us to make a framework selection based on a weak justification. Opting for choice c) means that the compiler does not need to be translated, nor are there any limitations on the choice of web -application framework. Furthermore, even if the compiler and the web-application were natively developed in the same language, choice c) still enforces the most robust separation of concerns.

Following this, we decompose the FunCompiler into two separate components: the web-application and the compiler. The web-application is the user-facing system, serving the web-pages and displaying the animations. The compiler behaves as a micro-service which receives a Fun program as input and outputs the result of the program's compilation (the contents of this result we will discuss more in Section 7.?).

Of course, the web-application must be able to send user-written Fun programs to the compiler. Likewise, the compiler must be able to return the result of the compilation to the web-application. Therefore, the two components require some method of communication. We do this by exposing the compiler as an API. The API under these circumstances is very simple and requires only one endpoint.

When a user enters a program within the web-application and chooses to animate either contextual analysis or code generation, the web-application makes an API request to the compiler, sending the input program. When the API receives this request, it passes the program as input to the compiler, executes the compilation and sends a response containing the output back to the web-application. The web-application uses the contents of this response to build the corresponding animation. Figure 6.2 illustrates this flow of information.

6.3 Technologies & Tools

With the FunCompiler split into its constituent components, we discuss the various system-level technologies and tools that will be used to implement, build and run the application. We start by examining the application-
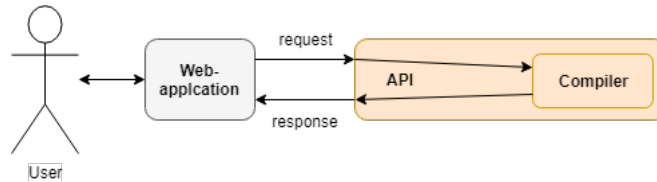
Figure 6.2: Web-application communicating with the compiler via an API

level technologies used by the web-application and the compiler individually, then we consider how to create an abstraction of the system as a whole and package it as a single, portable instance.

6.3.1 Web-application Technologies

Laravel

A convenient simplification of the FunCompiler is that the system requires no persistent storage, meaning our system architecture need not include a database. However, even without a database, the web-application is still required to perform a small amount of server-side (or back-end) duties, namely delivering API requests and handling the corresponding responses. We implement this functionality using Laravel, which is an open-source PHP framework intended for the rapid development of web-applications. Like the majority of web-frameworks, Laravel follows a standard *model-view-controller* (MVC) architectural pattern.

One might argue that given the limited amount of server-side processing required, a relatively large scale framework such as Laravel is excessive. Whilst it's true that a minimalistic framework (i.e., a Node framework such as Express) could achieve the same functionality, we note that under these circumstances, any performance difference between the two frameworks is likely to be negligible. Thus, we opt for Laravel due to the extensive experience the developer of this project has with the framework.

Using Laravel from the beginning also provides some sense of "future-proofing", in the sense that if the application were required to implement considerably more complex server-side functionality in the future, we already have a robust framework in place that is capable of handling these kind of complications, something that may not be possible with a minimalistic framework.

Composer

Within our PHP application we require several dependencies, including API libraries and Laravel itself. To manage and install these dependencies we use Composer, which is the quintessential dependency manager for PHP applications.

Composer utilises a configuration file called composer.json in which we can define all the PHP dependencies we require along with the *desired* versions of those dependencies. By executing `composer install`, Composer will download and install the dependencies specified within composer.json and generate composer.lock, which specifies the *actual* version of each dependency installed. After composer.lock has been generated, subsequent invocations of `composer install` will only download the versions of each dependency as specified in composer.lock, regardless of the version stated in composer.json.

This versioning system is practical because it means that all developers of a project will install the same versions of each dependency upon executing `composer install`, assuming composer.lock is committed

to some kind of version control system. If we do wish to update the dependencies to more recent versions, we invoke `composer update`, which updates **composer.lock** accordingly.

6.3.2 Compiler Technologies

ANTLR

The Fun language and the associated compiler were built using ANTLR, which is a tool that when given a grammar (a file defining the syntax of language), can automatically generate certain components of a standard compiler, such as a lexer and a recursive-descent parser. ANTLR also implements a visitor interface which provides the foundation for executing traversals over the parser-generated syntax tree, allowing the developer to implement a contextual analyser and code generator.

Spark

Alone, the compiler has no way of interacting with the web-application. To enable this, we form a channel of communication between the two by encapsulating the compiler within an API which contains a single endpoint. The endpoint receives a user submitted input program from the web-application, passes the program as input to the compiler, receives the output of the compilation and then passes this as a response back to the web-application.

Since the API requires only a single (and relatively simple) endpoint, we look to choose a framework that will allow us to expose an API with as little overhead as possible. The only limitation being that the framework must be Java-based in order to be compatible with the compiler. Consequently, we select a Java micro-framework ideal for creating APIs with minimal boilerplate called Spark (not to be confused with Apache Spark). Spark requires minimal setup and has extremely concise syntax, similar to that of functional languages, meaning we can create an API with very few lines of code (of which we will see in Section 7.?).

Maven

Maven is the definitive standard in regards to automated build tools for Java projects. The FunCompiler uses Maven to package the compiler and corresponding API into a single JAR file. Upon execution of this JAR file, the Spark server is launched which enables the compiler to accept API requests.

To configure a Maven project the developer uses a *Project Object Model* to define firstly how the software should be built, and secondly the dependencies it requires (conveniently, Maven has built-in dependencies for both Spark and ANTLR). This configuration is stored in an XML file named **pom.xml**. The contents of the FunCompiler's **pom.xml** can be found in Appendix A.1. With the Project Object Model defined, we can simply invoke `mvn package` to compile all the Java source files and package the deliverable code into an executable JAR file.

6.3.3 Docker

Given what has been previously discussed, our current system infrastructure is illustrated in Figure 6.3. We see how through the use of Composer and Maven we have already created convenient techniques for automating the dependency retrieval and build process of the web-application and compiler respectively. However, we now consider if we can find a tool that will allow us to apply this same technique to the project as a whole. In other

words, a tool which will automatically trigger the build, dependency retrieval and execution of both the web-application and the compiler at the same time. This is possible through a software known as Docker, which permits operating-system-level virtualisation.
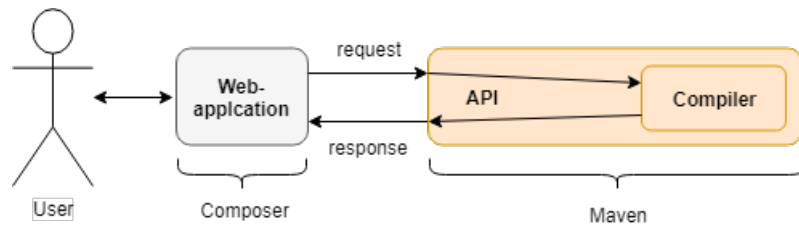

Figure 6.3: Current system infrastructure

Docker is designed to make it easier to create, deploy, and run applications by using ***containers***. A container can be thought of as a miniature, high-speed virtual machine. Prior to obtaining a container, we must first build an ***image***. To build an image we use a configuration file known as a ***Dockerfile*** in which one can specify commands that determine the environment of the image. These commands include downloading operating systems, setting file permissions, launching software, etc. After using a Dockerfile to build an image, we can then "run" it, which creates a corresponding live container, as illustrated in Figure 6.4. It may be helpful to think of a container as an instance of an image.
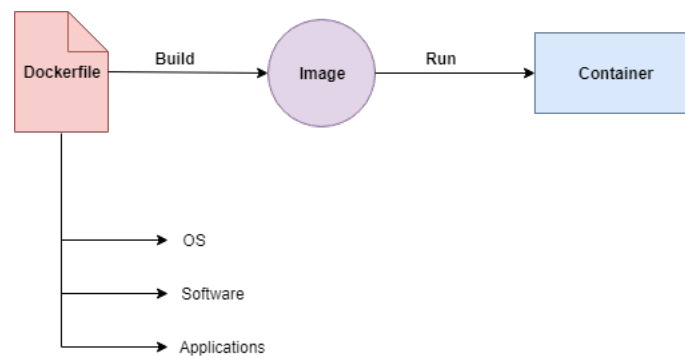

Figure 6.4: Creating a docker container from an image

The FunCompiler contains two Dockerfiles which are used to build two different images: one for the web-application, one for the compiler. Whilst we won't delve into specifics, these Dockerfiles contain the commands to retrieve dependencies (using Composer or Maven), execute builds and launch the servers. The exact contents of these Dockerfiles can be found in Appendices A.2 and A.3

We can then use a sub-tool of Docker for running multi-container applications called Docker Compose. we use a YAML file named docker-compose.yml to define services. Each service generally corresponds to one image and thus one Dockerfile. In our case, we have two services, the web-application and the compiler. By invoking `docker-compose up`, we launch the services defined in docker-compose.yml, which entails automatically building the images from the Dockerfiles and then running these images, which creates the containers. Once both containers are live, the application is effectively up and running. To see the full contents of the FunCompiler's docker-compose.yml file refer to Appendix A.4.

There are a litany of advantages to using Docker but the fundamental point is that if another developer (or perhaps the system admin of a server deploying the application) was to obtain the source code of the FunCompiler, aslong as they had Docker installed, they could install every dependency, execute all builds and enable the entire FunCompiler application without having any of the constituent software available locally (e.g., PHP, Composer, Maven, Apache, etc.). Therefore, this makes the FunCompiler extremely portable because every machine, regardless of its own architecture, will build the application in precisely the same way, with precisely the same versions of all dependencies.

With Docker implemented, we present the final system architecture of the FunCompiler, as demonstrated in Figure 6.5. We see that both the web-application and the compiler are represented as containers and are both operating and communicating within a Docker environment.
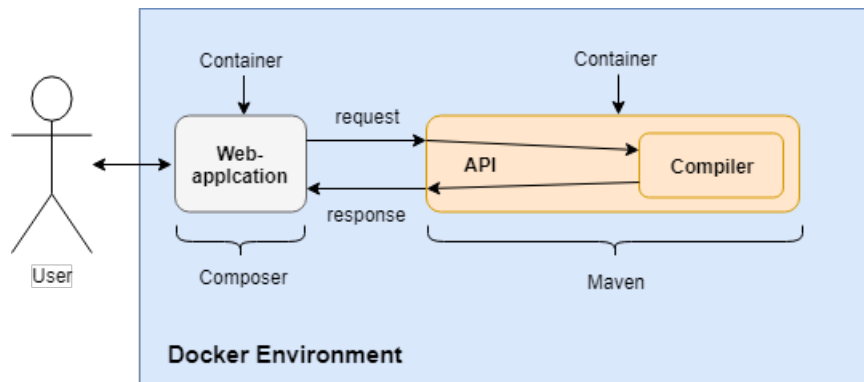

Figure 6.5: Final infrastructure of the FunCompiler

Chapter 7

Implementation

In this chapter we give a high-level overview of the FunCompiler's implementation. As in Chapter 6, we decompose the FunCompiler into two separate components: the web-application and the compiler.

Firstly, compiler. Secondly, web. Finally, tests

7.1 Compiler Implementation

In this section we discuss how we convert the original, stand-alone compiler into the micro-service as described in Chapter 6. In order to accomplish this, two fundamental changes are necessary. Firstly, we must expose the compiler as an API in order to enable communication between itself and the web-application. Secondly, we must modify the implementation of the compiler to return a large amount of structured data pertaining to the compilation of a particular program. It is this data that the web-application will receive and use to build an animation of either contextual analysis or code generation.

7.1.1 API Implementation

Recall from Section 6.3.2 that we intend to utilise a Java micro-framework known as Spark to encapsulate the compiler and create a compact API with a single endpoint. More specifically, the endpoint will be configured to accept HTTP POST requests. If the web-application desires to compile a Fun program, it must make a POST request to the API containing the following two query parameters: `program` and `type`. The value of `program` is simply the plain text of the input program. The value of `type` determines whether we wish to animate contextual analysis or code generation, in other words, the value of `type` corresponds to which of the buttons was pressed below the code editor in Figure 5.1. Listing 7.1 contains the implementation of the endpoint in its entirety.

Listing 7.1: API implementation

```
1  post("/", (req, res) -> {
2     res.type("application/json");
3     String program = req.queryParams("program");
4     String type = req.queryParams("type");
5     InputStream programInputStream = new ByteArrayInputStream(program.getBytes());
6     return FunRun.execute(programInputStream, type);
7  }, gson::toJson);
```

We see in the above listing how we utilise Java 8 lambda expressions to define a POST route at the url "/".
Within the body of the route we extract the values of the request's query parameters (`program` and `type`,
as previously discussed) and later pass them to the `execute()` method defined within the FunRun module.
`execute()` is the driver method in regards to the compilation of a program, and the return value of this method
is an object of the FunResponse class, of which each instance contains data collected pertaining to the compilation of an individual program. Finally, we utilise a Java library named GSON to convert the contents of the
FunResponse object into a set of JSON data, which is stored in the endpoint's response and then delivered back
to the calling entity.

7.1.2 Compiler Output & The FunResponse Class

As discussed, we are required to modify the compiler's implementation to enable the collection of various details
relevant to the compilation of a program, these details will be sent to the web-application and used to build an
animation of the compilation process. To accomplish this, we use a class named FunResponse. For each program
that we receive from the web-application, we instantiate a new object of this class. This object is populated as
the compilation proceeds, and is returned upon completion of the compilation. The information that we desire
to collect is stored within the instance variables of this object, the declarations of which are given in Listing 7.2
(with access modifiers removed).

Listing 7.2: FunResponse instance variables

```
1  int numSyntaxErrors;
2  int numContextualErrors;
3  List<String> syntaxErrors;
4  List<String> contextualErrors;
5  JsonArray treeNodes;
6  JsonArray nodeOrder;
```

In the interest of brevity we will not go into details of how or where the value of these variables is derived, however, we will analyse the contents of the variables and consider how they can be utilised by the web-application to build an animation.

Syntax & Contextual Errors

The first four instance variables of Listing 7.2 on lines 1-4 simply serve to store any syntactic/contextual errors
that the compiler detects within the input program. Lines 1 and 2 collect the number of errors, whereas lines 3
and 4 collect a list of the actual error messages, including line and character numbers.

It should be noted that the contents of these variables have no bearing on the form or behaviour of the
animation, their purpose is to provide a means of allowing the web-application to report errors to the user in
the event that the input program was invalid, and thus the animation not possible. In the case that the program
contains no errors, these variables will remain empty and unused.

AST Representation

The fifth instance variable, `treeNodes`, is used to store a textual representation of the AST which when delivered to the web-application, can be used to create and display a visual representation within the user interface.
The data stored within this variable is an array of JSON objects (forming a `JsonArray` data type), where each

object corresponds to a distinct node within the program's AST. Each object contains four fields of information used to describe the node it represents:

- An ID, uniquely identifying the node.

- A parent ID, identifying the node's parent.

- A node name, identifying the type of node (e.g, VAR, ID, PROC, etc.)

- A node value, identifying the text that is to be associated with the node when visually displayed within the AST (in many cases, this is the same as the node name).

By examining the ID and parent ID of each node, it is possible to take a *flat* representation of the AST, as we have in `treeNodes`, and interpret it as a *hierarchical* representation. For example, consider Listing 7.3 which contains an extract of three nodes from a typical `treeNodes` array.

Listing 7.3: Extract from the `treeNodes` array

```
{
  "id": 306481855,
  "nodeName": "DIV",
  "nodeValue": "DIV",
  "parentId": 1670278882
},
{
  "id": 254315796,
  "nodeName": "ID",
  "nodeValue": "n",
  "parentId": 306481855
},
{
  "id": 532745069,
  "nodeName": "NUM",
  "nodeValue": "2",
  "parentId": 306481855
}
```

Within this example we can see that we have three different types of node, a DIV, an ID and a NUM. By examining the ID of the DIV node and the parent ID of the ID and NUM nodes, it is possible to ascertain that ID and NUM are child nodes of DIV. By applying this knowledge, we can perceive the hierarchical relationships between the nodes of the AST through inspect of the flat array. As an illustration of this theory, in Figure 7.1 we present the visual representation of the AST as derived from the above node array.
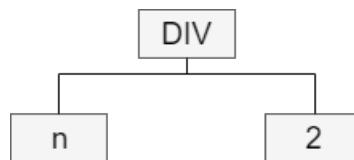

Figure 7.1: The visual representation of an AST as derived from the previous node array

Animation Order & Augmentations

Now that we have a means of building a visual representation of the AST via the `treeNodes` array, we still need some way of informing the web-application as to how the AST should be animated, i.e., in which order

32

each node should be highlighted. Also, each time a node is highlighted, we need to provide a set of data which can be parsed to build and display the relevant augmentations, as discussed in Chapter 5. We do this through the use of a final instance variable, `nodeOrder`. Similarly to `treeNodes`, `nodeOrder` contains an array of JSON objects. For example, consider Listing 7.4 which contains an extract of three JSON objects from the `nodeOrder` array:

Listing 7.4: Extract from the `nodeOrder` array

```
{
  "id": 306481855,
  "explanations": [
    "Walk expr1"
  ],
  "table": [
    {
      "scope": "global",
      "id": "n",
      "type_address": "int"
    }
  ]
},
{
  "id": 254315796,
  "explanations": [
    "Lookup 'n' and retrieve its type, int"
  ],
  "table": [
    {
      "scope": "global",
      "id": "n",
      "type_address": "int"
    }
  ]
},
{
  "id": 306481855,
  "explanations": [
    "Walk expr1",
    "Walk expr2"
  ],
  "table": [
    {
      "scope": "global",
      "id": "n",
      "type_address": "int"
    }
  ]
}
```

The important thing to recognise is that each object within the `nodeOrder` array has a direct link to a node in the `treeNodes` array via the ID field. In the above example we see that the first and third object have the same ID as the `DIV` node in Listing 7.3, and that the second object has the same ID as the `ID` node. The relevancy of these links is that the order of the nodes in `nodeOrder` dictates the order in which the nodes in `treeNodes` should be highlighted during the animation. Applying this logic to the Listings in 7.3 and 7.4 indicates that the animation should first highlight the `DIV` node, then the `ID` node, and finally the `DIV` node once again, as illustrated in Figure 7.2.
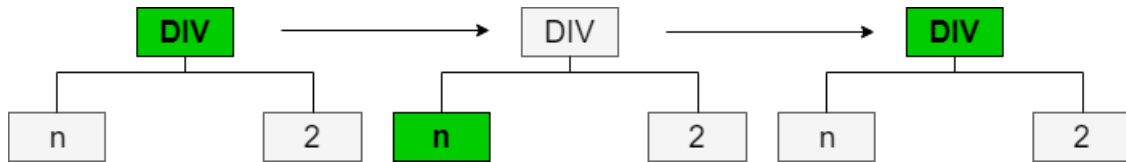
Figure 7.2: The animation of the AST as dictated by the Listings 7.3 and 7.4

We also see that we have fields for "explanations" and "table". These fields represent some of the augmentations that were discussed in Chapter 5. When the node in the AST represented by the ID field is highlighted, the animator simultaneously uses the value of these fields to display the appropriate augmentations alongside the animation. The sort of augmentations that are supplied to the `nodeOrder` array are dependent upon the value of the query parameter `type`, as discussed in Section 7.1.1. Figure 7.3 extends the animation shown in Figure 7.2 by inserting the "explanations" augmentation as specified in Listing 7.4.
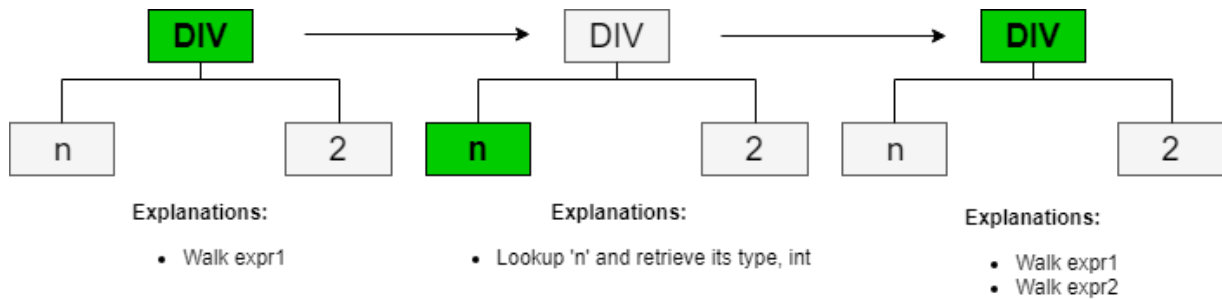
Figure 7.3: The animation of the AST as dictated by the Listings 7.3 and 7.4 with explanatory messages added

Within the `nodeOrder` array we have effectively stored a complete history of the program's compilation. That is, for each individual time a node is visited, we have retained a "snapshot" of any relevant information at that exact moment in time (e.g., explanatory messages, type table, etc.). Since the animator can freely traverse forwards or backwards through the `nodeOrder` array whilst maintaining all the correct information, we can almost trivially implement the playing, restarting and reversing of an animation as discussed in Chapter 5.

7.2 Web-application Implementation

In this section we examine the physical implementation of the interface designs we created in Chapter 5. For each component in the interface we detail its purpose and functionality along with any interesting technologies or libraries that were used to assist in its develop. Also note that, although not discussed specifically, the front-end implementation of this project does make extensive use of HTML, CSS, Bootstrap, vanilla Javascript, jQuery and NodeJS.

7.2.1 Code Editor Implementation

Firstly, we consider how to implement the code editor as designed in Section 5.1.1. Within this section we remarked how, ideally, the code editor should follow some of the semantics of standard programmatic text editors. To produce these semantics, we use a Javascript library known as CodeMirror which allows the embedding of a versatile text editor within a web application. CodeMirror provides support for bracket matching, auto-indentation, syntax highlighting for over 100 predefined languages and much more.

Unsurprisingly, the Fun language is not included as one of the predefined languages that is provided built-in syntax highlighting, however, CodeMirror does allow the developer to define a custom lexer for non-standard languages. We create a simple lexer for Fun in which we state some of the language's syntactic details such as keywords, types and identifiers. The source code for this lexer is provided in Appendix B.1.

The implementation of the code editor is provided in Figure 7.4. We can see how syntax highlighting has been provided for the various different syntactic structures along with line numbers, line highlighting, and whilst not visible from the diagram, auto-indentation. Below the input code two buttons have been inserted, one for contextual analysis and the other for code generation. Pressing one of these buttons triggers the web-application to send an API request to compiler. The program currently entered in the code editor and the button pressed form the values of the query parameters `program` and `type` as discussed in Section 7.1.1.
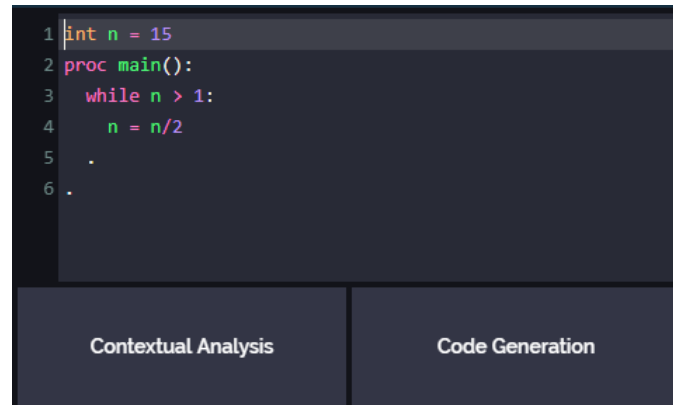
Figure 7.4: Implementation of the code editor

7.2.2 AST Implementation

In Section 5.1.2 we discussed how we require the web-application to display an AST in the conventional hierarchical tree format, alongside highlighting nodes consecutively in order to create an animation. We accomplish this using another Javascript library named D3, which is a popular tool for data visualisation.

After pressing one of the buttons in Figure 7.4 and delivering an API request to the compiler, we can use the information contained within the returned response to build the AST. In the same way that we interpreted Listing 7.3 to build Figure 7.1, we can use D3 to parse the `treeNodes` array contained within the API response to build a data structure containing a hierarchical data structure representing the AST. With this data structure created, D3 can automatically generate an SVG representation of the AST. In Figure 7.5 we illustrate a series of different ASTs generated from different Fun programs using D3. We see how regardless of the number of the nodes within the AST, D3 can intuitively shrink or condense the layout of the tree to create the most comprehensible visualisation possible.

We can also see in 7.5 how the `PROG` node of each AST has been expanded and coloured blue, this is how we will highlight each node within the AST during the animation. To implement the animation in the correct order recall we simply need to iterate through th `nodeOrder` array, also contained within the API response. When we use D3 to build the SVG of the AST, we associate with each node in the AST its ID as specified in the `treeNodes` array. This means when attempting to determine which node in the AST to highlight next, we can look at the next ID specified in `nodeOrder` and highlight the node in the SVG with that ID. Figure 7.6 illustrates how the animation of the AST will look as we highlight nodes consecutively.
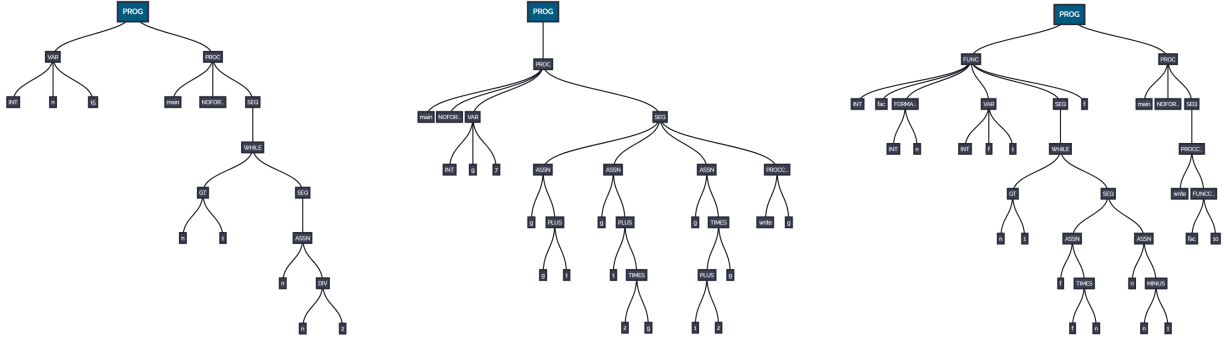
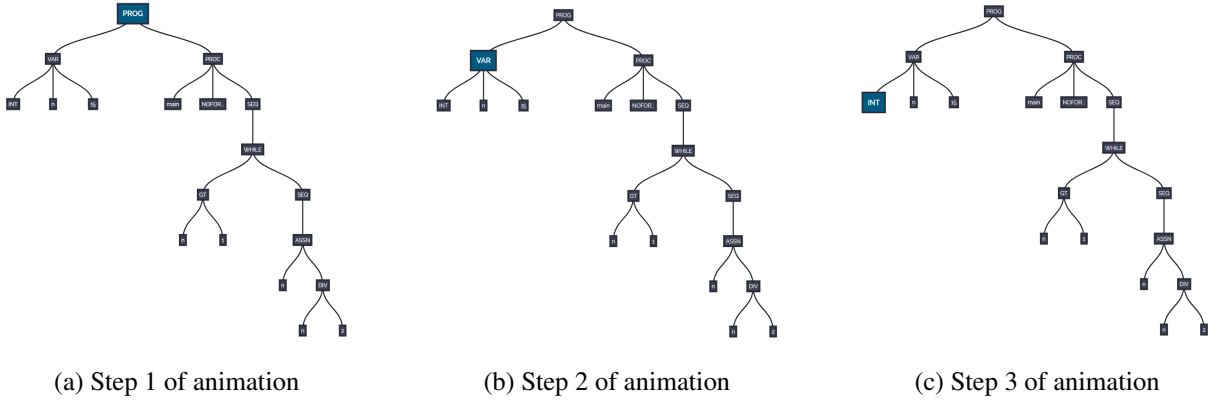Figure 7.5: D3 generated ASTs from three different Fun programs

(a) Step 1 of animation   (b) Step 2 of animation   (c) Step 3 of animation

Figure 7.6: The first three steps of a Fun program's animation

7.2.3 Augmentations Implementation

In Section 5.1.3 we created designs for the augmentations that would accompany each step of the animation. During each step of the animation and as each node is highlighted, the web-application retrieves a list of augmentations from the `nodeOrder` array, creates visual representations, and displays them alongside the animation. We see the implementation of the augmentations for both contextual analysis and code generation in Figure 7.7.

In addition to displaying the corresponding augmentations for each type of animation, we have also included the name of each node being animated along with highlighting the line in the explanatory messages section that describes the current action of the compiler.

7.2.4 Playback Controls Implementation

To implement the playback controls designed in Section 5.1.4 we simply embed playback Bootstrap "glyphicons" above the AST, as illustrated in Figure 7.8. Note that we have also included the title of the current compilation phase being animated.

7.2.5 Fun Specification Implementation

The final interface component we are required to implement is the Fun specification as designed in Section 5.1.5. This implementation of this is demonstrated in Figure 7.9 which demonstrates how we can navigate through each

Node: GT

Code Checker Actions

> Walk expr1
> Walk expr2
> Check both expressions have type int
> Check the first expression is of type int
> Success, type is int
> Check the second expression is of type int
> Success, type is int

Type Table

Scope	ID	Type
global	read	void -> int
global	write	int -> void
global	n	int
local	main	void -> void

(a) Contextual analysis augmentations

Node: PROG

Code Generator Actions

> Predefine the read and write procedures
> Walk var-decl, generating code
> Note the current instruction address, c1 (3)
> Emit 'CALL 0'
> Emit 'HALT'
> Walk proc-decl, generating code

Code Template

> Code to evaluate variable declarations
> CALL
> HALT
> Code to evaluate procedure declarations

Object Code

0: LOADC 15
3: CALL 0
6: HALT

Address Table

Scope	ID	Address
code	read	32766
code	write	32767
global	n	0

(b) Code generation augmentations

Figure 7.7: Implementation of the augmentations

tab and view different sections of the specification.

7.2.6 Complete Implementation

Finally, we can combine each of these interface implementations to create complete designs. Figure 7.10 illustrates the full view of the landing page, the contextual analysis animation page and the code generation animation page. Note that we have added a "home" button within the navigation menu to return to the landing page along with a drop-down menu of "examples". These examples contain sample Fun programs that when clicked, automatically fill the code editor.

7.3 Testing

The FunCompiler contains a fairly extensive test suite that mainly uses PHPUnit to test that the compiler API returns the correct information. The FunCompiler is also configured to use Continuous Integration via Travis. Each time a change a made to the FunCompiler and pushed to version control, the test suite is run through Travis.
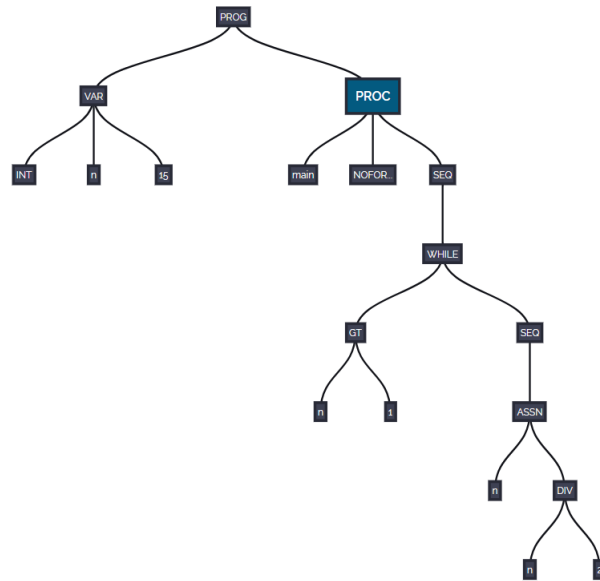
Figure 7.8: Implementation of the code editor

Figure 7.9: Implementation of the Fun specification

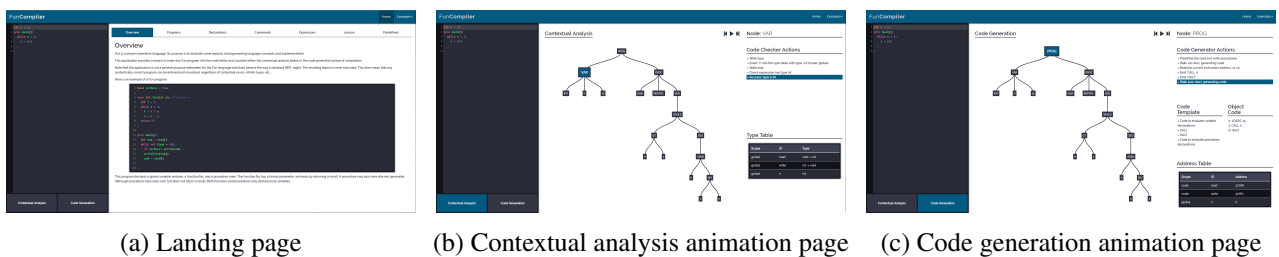(a) Landing page   (b) Contextual analysis animation page   (c) Code generation animation page

Figure 7.10: The complete implementation of each of the FunCompiler

Chapter 8

Evaluation

Chapter 9

Conclusion

Appendices

Appendix A

Configuration Files

A.1 pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>api</groupId>
  <artifactId>api</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>com.sparkjava</groupId>
      <artifactId>spark-core</artifactId>
      <version>2.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.7.21</version>
    </dependency>
    <dependency>
      <groupId>org.antlr</groupId>
      <artifactId>antlr4-runtime</artifactId>
      <version>4.5.3</version>
    </dependency>
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.0</version>
    </dependency>
  </dependencies>
```

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <finalName>api</finalName>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <mainClass>api.Api</mainClass>
            <classpathPrefix>dependency-jars/</classpathPrefix>
          </manifest>
        </archive>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>attached</goal>
          </goals>
          <phase>package</phase>
          <configuration>
            <finalName>api</finalName>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
            <archive>
              <manifest>
                <mainClass>api.Api</mainClass>
              </manifest>
            </archive>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
```

```
          <groupId>org.antlr</groupId>
          <artifactId>antlr4-maven-plugin</artifactId>
          <version>4.5.3</version>
          <executions>
            <execution>
              <goals>
                <goal>antlr4</goal>
              </goals>
              <configuration>
                <sourceDirectory>${basedir}/src/main/antlr4/api</sourceDirectory>
                <listener>false</listener>
                <visitor>true</visitor>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>

</project>
```

A.2 Web-Application Dockerfile

```
# Use a PHP7.0 image that is packaged with the Apache web server
FROM php:7.0-apache
# Download composer and install composer.phar in /usr/local/bin
RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bi
# Create the 'app' directory and set to be the working directory of the container
WORKDIR /app
# Copy all files in current directory to the working directory of the container
COPY . /app
# Update packages and install composer and PHP dependencies.
RUN apt-get update -yqq
RUN apt-get install git zlib1g-dev -yqq
RUN pecl install xdebug
# Compile PHP, include these extensions.
RUN docker-php-ext-install zip
RUN docker-php-ext-enable xdebug
# Use the 'composer.json' file to download any PHP dependencies (including Laravel)
RUN composer install
RUN cp .env.example .env
RUN php artisan key:generate
```

A.3 Compiler Dockerfile

```
# Use an OpenJDK8 image to run and compile Java files
FROM openjdk:8
# Install Maven in the container
RUN apt-get update
RUN apt-get install -y maven
```

```
# Create the 'compiler' directory and set to be the working directory of the contai
WORKDIR /compiler
# Add 'pom.xml' to the container
ADD pom.xml /compiler/pom.xml
# Download maven dependencies (e.g., Antlr and Spark)
RUN ["mvn", "dependency:resolve"]
RUN ["mvn", "verify"]
# Add the source directory
ADD src /compiler/src
# Compile and package into a fat jar
RUN ["mvn", "package"]
```

A.4   docker-compose.production.yml

```
# Version of the docker-compose file format
version: '3'

# Individual components of the overall product (the web app and the compiler api)
services:
    compiler:
        # Name the resulting container
        container_name: FunCompiler
        # Configuration options applied at build time (i.e., location of Dockerfile
        build: ./compiler
        # Launch the Spark web server (the compiler API)
        command: java -jar target/api-jar-with-dependencies.jar
        # Expose port 8001 on the host and 4567 on the container
        ports:
            - 8001:4567

    app:
        # Name the resulting container
        container_name: FunApp
        # Configuration options applied at build time (i.e., location of Dockerfile
        build: ./app
        # Launch the Laravel server (the web app)
        command: php artisan serve --host=0.0.0.0 --port=8000
        # Expose port 8000 on the host and port 8000 on the container
        ports:
            - 8000:8000
        # Starts services in dependency order, i.e., compiler before app
        depends_on:
            - compiler
```

Appendix B

Change Me

B.1 CodeMirror Custom Lexer

```
CodeMirror.defineSimpleMode("fun", {
  start: [
    {regex: /(?:func|proc|return|if|while|else|not)\b/, token: "keyword"},
    {regex: /true|false/, token: "atom"},
    {regex: /int|bool/, token: "type"},
    {regex: /0x[a-f\d]+|[-+]?(?:\.\d+|\d+\.?\d*)(?:e[-+]?\d+)?/i,
        token: "number"},
    {regex: /#.*/, token: "comment"},
    {regex: /[-+\/*=<>]+/, token: "operator"},
    {regex: /[\:]/, indent: true},
    {regex: /[\.]/, dedent: true},
    {regex: /[a-z$][\w$]*/, token: "variable"},
  ],
  comment: []
});
```

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.