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

1 Introduction **1**

1.1 Motivation 1

1.2 Aims 2

1.3 Outline 3

2 Background & Related Work **4**

2.1 Effectiveness of Algorithm Animation 4

2.2 Existing Products 5

2.3 Discussion 6

3 The Fun Programming Language & Compilation Theory **8**

3.1 The Fun Programming Language 8

3.2 Syntax Trees 9

3.3 Compilation Phases 10

3.3.1 Syntactic Analysis 10

3.3.2 Contextual Analysis 11

3.3.3 Code Generation 12

3.4 Code Templates 13

4 Requirements **14**

4.1 Methodology 14

4.2 User Stories 15

4.3 Functional Requirements 15

4.4 Non-functional Requirements 17

4.5 Analysis 17

5 Design **18**

5.1 Application Interface 18

5.1.1 Code Editor 18

5.1.2 AST 19

5.1.3 Augmentations 19

5.1.4 Playback Controls 20

5.1.5 Fun Specification 20

5.2 Full Design 21

5.2.1 Landing Page 21

5.2.2 Contextual Analysis Animation Page 21

5.2.3 Code Generation Animation Page 22

Appendices **23**

Chapter 1

Introduction

With increasing technological advances and furthering levels of software abstraction, some may consider compilation to be a slightly esoteric subject, in that, broad knowledge of compilation theory is unnecessary in a modern environment and required only in a small number of highly specialised industries. Yet, remaining as a cornerstone of a computer science curriculum at many schools and universities is the art of compiler construction, behaviour and optimisation. 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 notion of computer science, upon which a larger syllabus can be developed. To students, the benefits of learning about compilation are perhaps not in its direct application, but in its ability to act as the stepping stone that facilitates a more comprehensive grasp of other concepts; concepts which themselves are grounded in compilation theory.

1.1 Motivation

Compilation can often be a challenging field to teach effectively. Many components 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 demonstrate. We desire to find a solution to this problem by creating techniques that provide concrete representations of abstract ideas.

In response to this issue, a common approach employed by educators is to 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. In Figure 1.1 we see that each consecutive slide illustrates the next step of the traversal by highlighting the corresponding node.

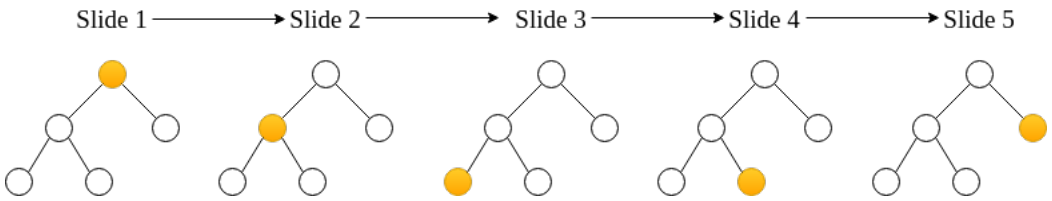


Figure 1.1: A sequence of slides demonstrating a depth-first traversal over a binary tree

Whilst using a slideshow to demonstrate these concepts does provide part of the solution to representing abstract notions, they have several limitations. Firstly, to create visualisations in this way is an arduous task on behalf of the educator, realistically meaning that 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 available 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, there is effectively an infinite number of ways the compilation process may occur depending on the input to the compiler. 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 section of code. Fun is a simple, educational programming language equipped with a compiler. Chapter 3 introduces all of these concepts in considerably more detail.

The application will allow users to supply programs 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 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, similar to the method displayed in Figure 1.1. In general, the application should:

- Allow a user to input any syntactically valid 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).

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

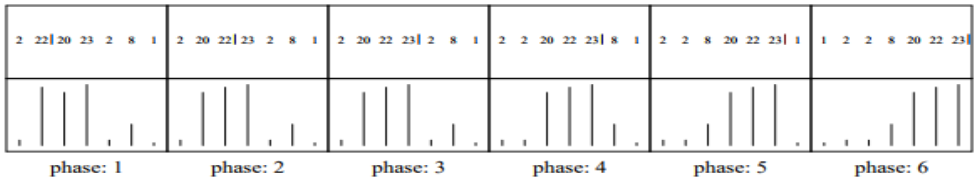


Figure 2.1: A sequence of stills from an insertion sort algorithm

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:

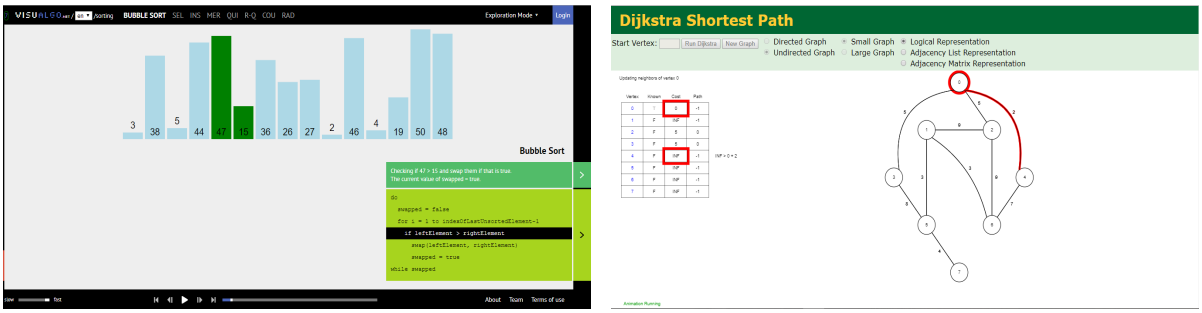
- The animation should be augmented with textual descriptions.
- The animation should include rewind-replay capabilities.
- Students should be able to build the animation themselves.

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 an 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 phases of compilation, 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 or assembly language 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". PL/0 was intended as a small educational programming language, used to teach the concepts of compiler construction. The language contains very primitive constructs and limited operations. Similarly to PL/0, "Fun" is a simple imperative language built using ANTLR [1], developed at Glasgow University by David Watt and later extended by Simon Gay. Its purpose is to 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 Fun program which utilises variables, functions and procedures to calculate and output the factorial value of user input:

```
bool verbose = true

func int fac (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() .
.
```

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 a function/procedure 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 `fac`. All variables in Fun have a size of 1.

Whilst Fun may differ significantly to 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.

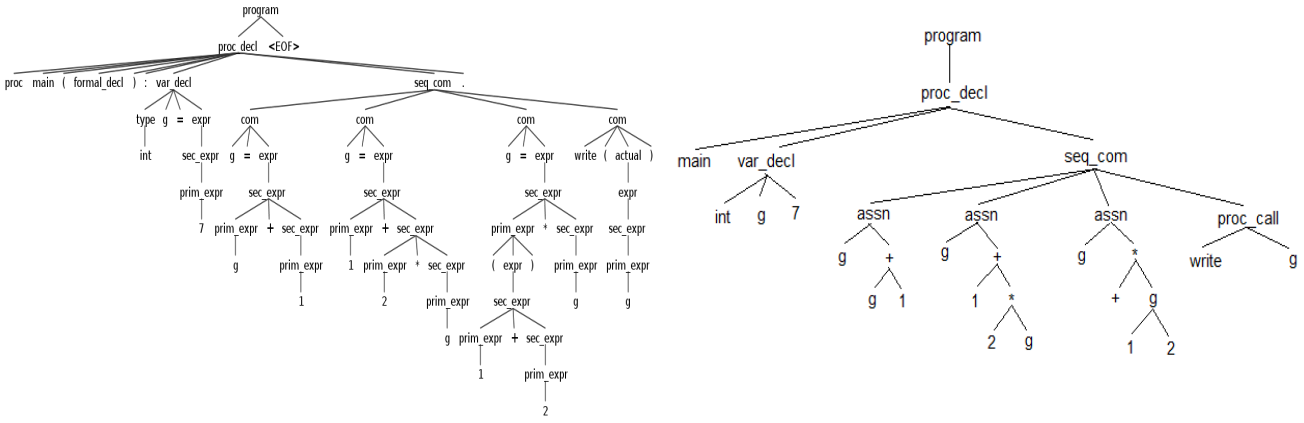
3.2 Syntax Trees

During compilation, a source program is parsed into a ***syntax tree***. A syntax tree is a hierarchical syntactic representation of a source program; with global statements towards the root, and more deeply nested statements towards 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 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 typically very 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



(a) Parse tree

(b) Abstract syntax tree

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

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.

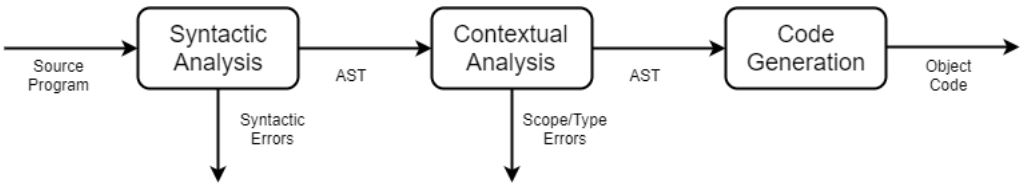


Figure 3.2: Compilation Pipeline

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

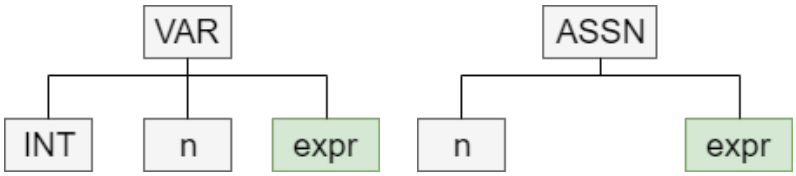
Upon successful completion of syntactic analysis, the generated AST is traversed by a contextual analyser. A contextual analyser checks whether the source program represented by the AST conforms to the source language's scope and type rules. Contextual analysis utilises an auxiliary data structure called 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 snippets of an AST 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: AST snippets demonstrating the declaration and usage of a variable `n`

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 to determine its type. The contextual analyser will then check that the type of the expression is the same as the type of `n`. For example, given that `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 a variable declaration as it is traversing over the AST, it determines a suitable address and inserts that value along with the corresponding identifier and scope into the address table. In the case of Fun, since all variables have a size of 1, the compiler can simply allocate incrementally consecutive addresses to each variable as it is encountered.

Code selection selects and generates the 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, which will be discussed in the next section.

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 instructions (written in the plain English) that defines how any constituent expressions/commands of the construct should themselves be considered for translation. The template also establishes specifically which assembly language instructions to emit.

For example, in Figure 3.7 we see an AST snippet that demonstrates the addition of two expressions, where `expr1` and `expr2` are sub-expressions of the `PLUS` construct. Consider the case where `expr1` is a variable, say `x`. In order to ensure the value of `x` is available in memory when the addition occurs, the compiler needs to emit the object code which loads the value of `x` into memory before emitting the addition object code. Naturally, the same logic applies to `expr2`, regardless of whether or not it is in the form of a variable or just a constant. Therefore, the code template for the `PLUS` construct should state that `expr1` and `expr2` should be evaluated before we emit any object code to add the expressions together.

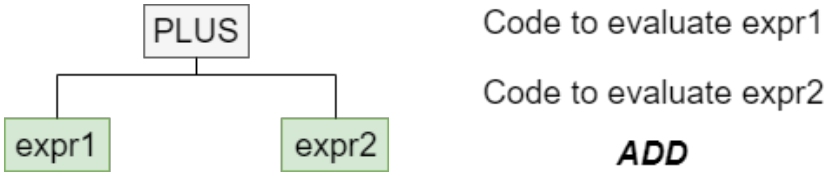


Figure 3.7: AST snippet and code template of a `PLUS` construct

Figure 3.7 also illustrates a code template for the `PLUS` construct. Considering each instruction sequentially, it stipulates that firstly we should generate the object code for `expr1`, then `expr2`. Only after both these values have been loaded into memory can we emit the `ADD` object code instruction.

It is often the case that the compiler needs to translate the constituent statements of a construct into object code before emitting any object code for the construct itself. In general, how each construct should be translated into object code is not always immediately clear just from the structure of the AST. Code templates assist in solving this matter 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**).

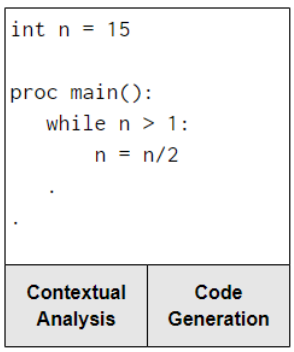
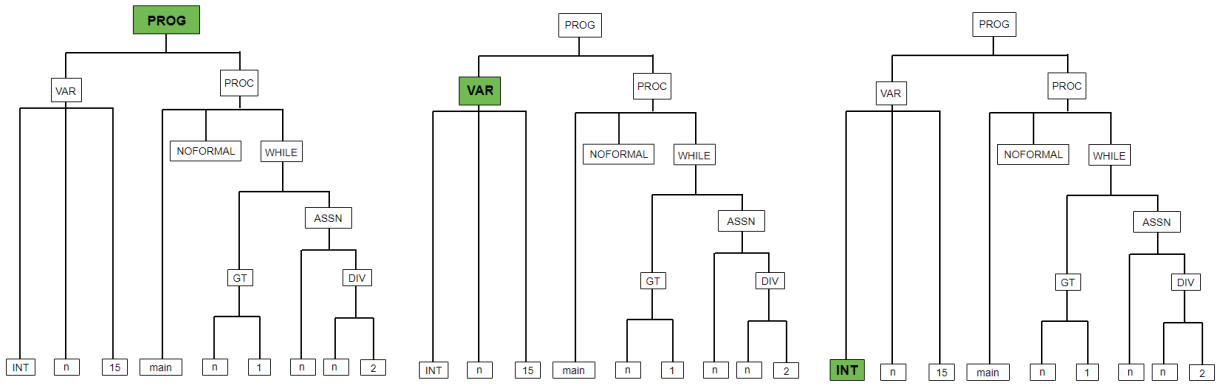


Figure 5.1: Code editor wireframe

5.1.2 AST

The AST of the input program is the foundation of the animation. The AST will be displayed in a 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.

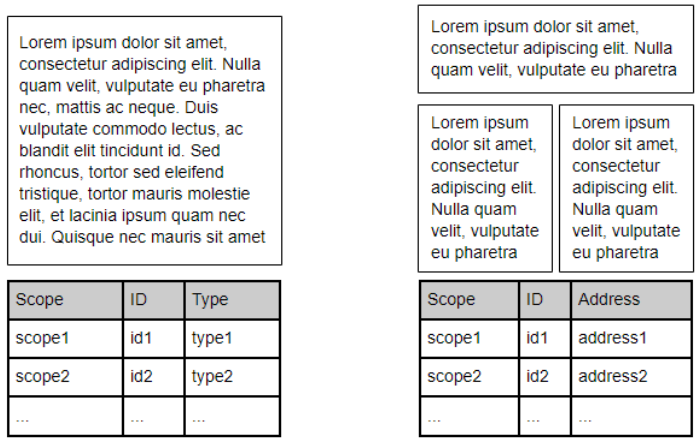


(a) Step 1 of animation (b) Step 2 of animation (c) Step 3 of animation

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 watcher's understanding, 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**).



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

Figure 5.3: Augmentations wireframe

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 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**).

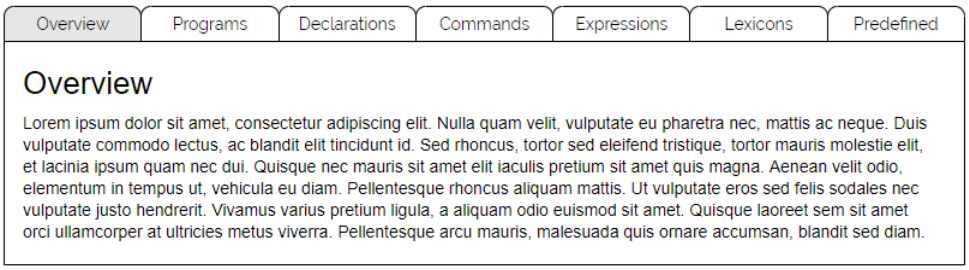


Figure 5.5: Fun specification wireframe

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 a naturally inherent link between these three components. For example, when users generate an AST, it is often necessary to cross-reference between the AST and input code that caused its creation, therefore the code editor with the preserved code must remain in view during the animation. 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.

Additionally, we must ensure we reserve sufficient space to embed the Fun specification. We achieve the above by splitting the FunCompiler 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 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. Every page within the FunCompiler has access to the navigation bar seen at the top of the diagram, clicking the "FunCompiler" text within this bar will return the user to the landing page.

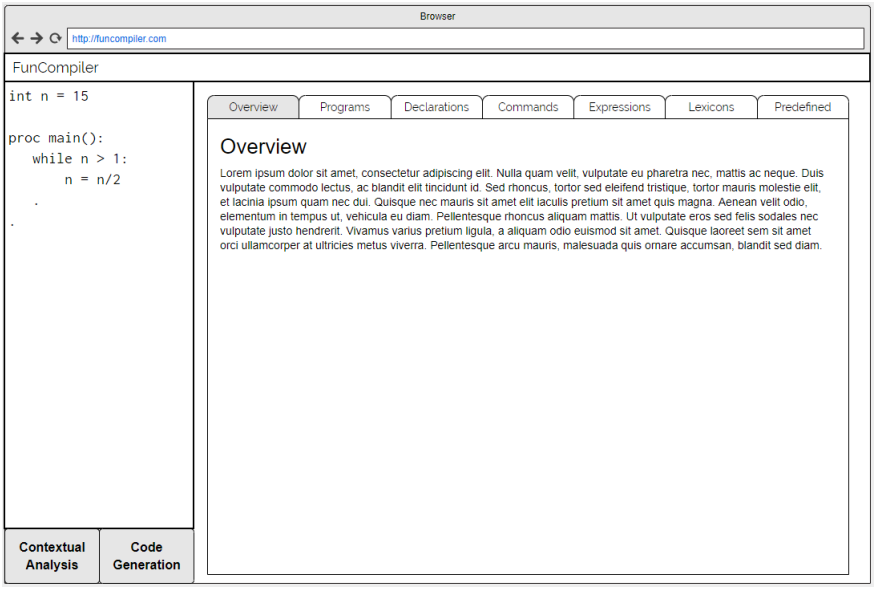


Figure 5.6: Landing page wireframe

5.2.2 Contextual Analysis Animation Page

If the user chooses contextual analysis, they are shown the page illustrated in Figure 5.7. This page displays the AST for the input program, playback controls for the animation, a section to list messages explaining the compiler's logic and the type table. From here the user still has access to the code editor and is free to modify the input program and generate a new animation.

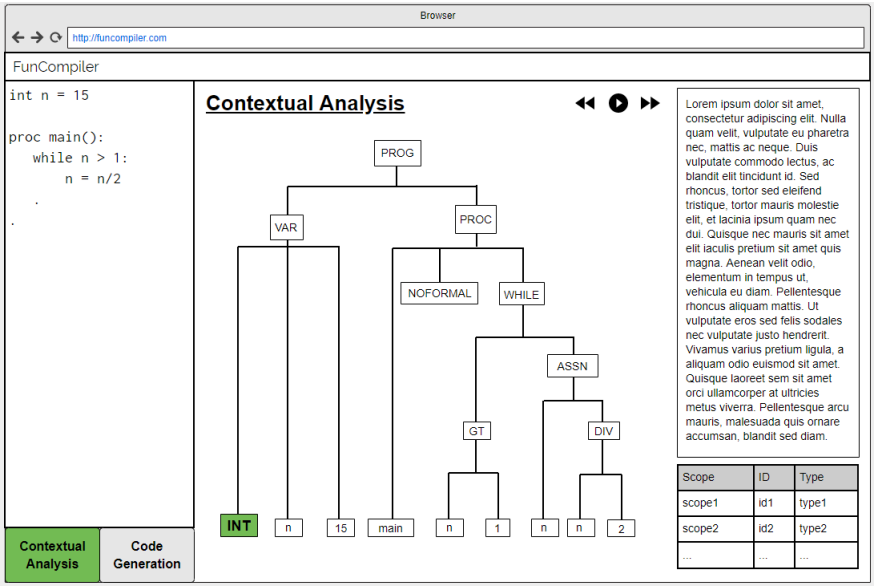


Figure 5.7: Contextual analysis page wireframe

5.2.3 Code Generation Animation Page

If the user chooses to animate code generation, they are shown 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.

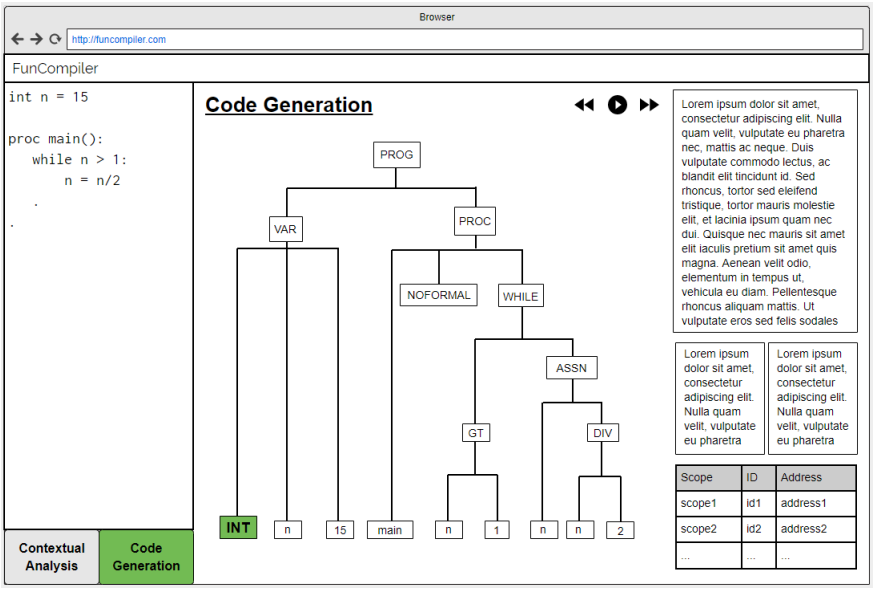


Figure 5.8: Code generation page wireframe

Appendices

Bibliography

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

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

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

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

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

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

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