University of Glasgow | School of Computing Science

Animation of the Contextual Analysis and Code Generation Phases of a Compiler

David Robertson - 2135967R

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2018

Abstract

The goal of this project is to create an application that provides concrete representations of the abstract notions characterised by compilation theory. The application, named the FunCompiler, is to be distributed as an additional educational resource during the delivery of the University of Glasgow's computer science course: *Programming Languages*. The FunCompiler enables its users to animate the compilation of any program written in the Fun programming language, this includes building visualisations of the key data structures and augmenting each step of the animation with supplementary, explanatory details. We also consider past research into the effectiveness of using animation to teach computational algorithms and attempt to use the FunCompiler to support the theory that algorithm animation is, in particular, more beneficial to those revising than those learning for the first time.

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*" [20]. 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 provide 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 slideshow contains 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.

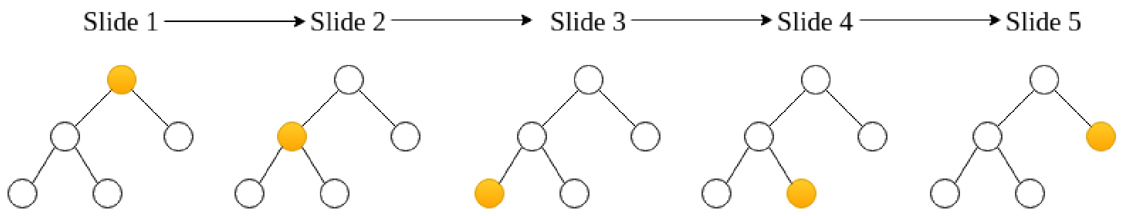


Figure 1.1: A sequence of slides demonstrating a depth-first traversal over a binary tree

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 predetermined 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 predetermined 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 which will be distributed as an additional educational resource during the University of Glasgow's level 3 computer science course: *Programming Languages*. The application will allow users 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 program. 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 the user. 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 any relevant supplementary information that may provide insight into the internal logic of the compiler.
- Allow the animation to be played, paused or stepped through (backwards and forwards).

The application resolves many of the problems we previously discussed with respect to slideshow-based visualisations. Most obviously, educators are no longer required to invest large amounts of time into creating the visualisations themselves. 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 regarding 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 predetermined input is eliminated. Finally, the application desires to augment the animation by automatically displaying informative details describing the compiler's progress and internal logic, 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 when teaching the topic.

1.3 Outline

The remainder of this dissertation is organised as follows:

Chapter 2 - **Background & Related Work**

Considers prior research that investigates the effectiveness of using algorithm animations in an educational manner. Additionally, explores and critiques any existing web-based algorithm animation tools.

Chapter 3 - **The Fun Programming Language & Compilation Theory**

Introduces the reader to the main theoretical concepts used throughout this dissertation.

Chapter 4 - **Requirements**

Establishes the requirements of the application and the methods by which they were elicited.

Chapter 5 - **Design**

Analyses the application's requirements to build an aesthetic design of the user interface.

Chapter 6 - **System Architecture**

Defines the technical infrastructure of the application and explains the rationale behind particular language and tool choices.

Chapter 7 - **Implementation**

Describes the physical implementation details of the application's key features.

Chapter 8 - **Evaluation**

Discusses the evaluations conducted to assess the quality of the application from multiple perspectives.

Chapter 9 - **Conclusion**

Summarises the project by reflecting on the lessons learned during development and suggests interesting possibilities for future work.

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 [3]. 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 learned 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 previous research 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 animation tool.

2.1 Effectiveness of Algorithm Animation

One of the earliest algorithm animation systems was developed by Bentley and Kernighan in 1987 [3]. 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 quite 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, in 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 animation assisted learning as much as we might think [15]. Stasko, Badre and Lewis carried out a study which involved testing two separate groups of students on their ability to learn the concept of a "pairing heap" [16], with one group using textual descriptions of the algorithm and the other using an animation of the algorithm.

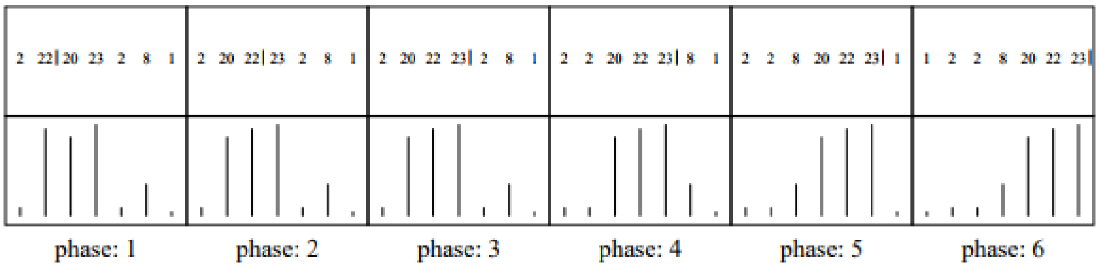


Figure 2.1: A sequence of stills from an insertion sort algorithm

Stasko then repeated a similar experiment in 1999 with Byrne and Catrambone [4]. 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. 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:

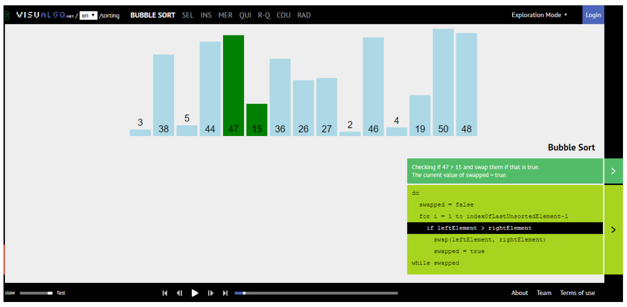
- The animation should be augmented with textual descriptions.
- The animation should include rewind-replay capabilities.
- Students should be able to build the animation themselves.

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. 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 ASTs) and virtually no tools exist that create animations of the compilation process as a whole. There is however, an abundance of web-applications that animate more straightforward algorithms, such as sorting or searching algorithms.

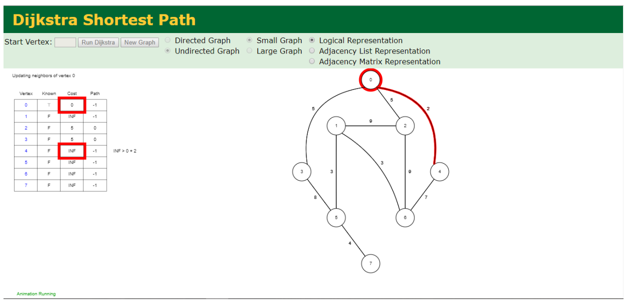
One of the most popular and well implemented is VisuAlgo [18]. VisuAlgo provides an interface for animating various sorting algorithms, including bubble sort, selection sort, etc. As shown in Figure 2.2a, VisuAlgo implements many of the guidelines we previously listed. It provides impressive playback 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.



Checking if 47 > 15 and swap them if that is true.
The current value of swapped = true.

```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true
while swapped
```

(a) VisuAlgo



Start Vertex: Undirected Graph, Small Graph, Logical Representation

Updating neighbors of vertex 0

Vertex	Known	Cost	Path
0	T	0	-1
1	F	INF	-1
2	F	5	0
3	F	5	0
4	F	INF	-1
5	F	INF	-1
6	F	INF	-1
7	F	INF	-1

INF > 0 + 2

(b) USF's Dijkstra's algorithm animation

Figure 2.2: Examples of existing algorithm animation software

Departing from pure sorting algorithms, the University of San Francisco (USF) has developed a small web-application which animates Dijkstra's algorithm [11], 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. The user cannot define their own input and they 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 of the algorithm's progress.

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 avoided if the environment in which an algorithm animation tool was targeted for use was selected more carefully. The studies suggest that, ideally, students should have been taught the concepts using conventional methods to begin with, then the educator should distribute the application to students who could utilise this as a means to refine and clarify their understanding. Fortunately, this describes precisely the scenario in which the application for this project would be used, where students of the Programming Languages course would be directed to use the tool as a secondary learning resource after first learning the concepts in a classroom setting.

We also consider the previous guidelines proposed by Stasko, Badre and Lewis and how they might apply to an application that animates a compiler. It appears the potential system should:

- Provide supplementary textual explanations that rationalise and justify the logic of the compiler.
- 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 compiler.

Finally, after analysing some examples of current products that are available in the area of algorithm animation, such as VisuAlgo and USF's Dijkstra's algorithm animator, it seems there is a trade-off between complexity and usability. In order to animate a more complex algorithm, USF sacrifices much of the visual support and playback control that VisuAlgo is able to provide.

Whilst we look to VisuAlgo as the ideal benchmark to strive towards in terms of the functionality it offers, we note that a compiler animator must overcome this trade-off in order to be educationally effective. That is, whilst animating an algorithm considerably more intricate than even Dijkstra's algorithm, we must also 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 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 1976 book *Algorithms + Data Structures = Programs* [19], 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 the University of Glasgow 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 used to help teach many of the concepts within the level 3 computer science course, Programming Languages, at the University of Glasgow.

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'`, where `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***, meaning 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`.

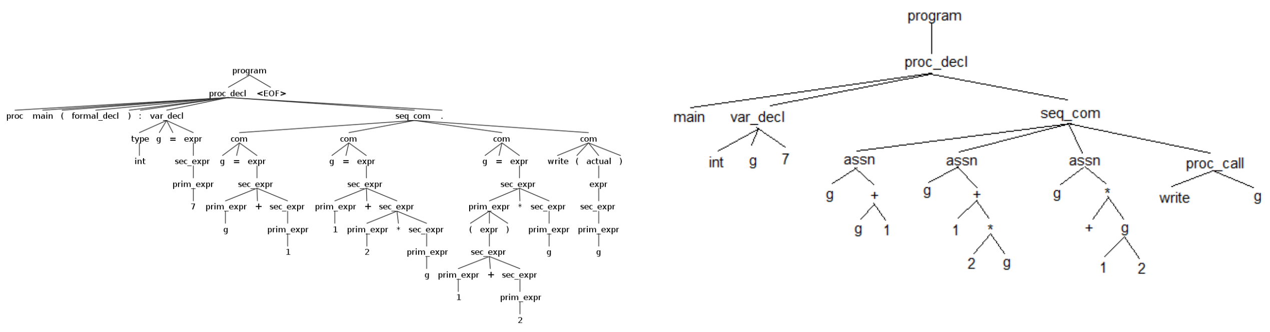
Whilst Fun may differ significantly from other programming languages, particularly in its complexity, it is not the case that the notions it aims to represent are exclusive to the Fun language only. Fun is sufficiently generic that core concepts can be delivered in a simple, comprehensible format (due 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: a ***parse tree*** (sometimes called a concrete syntax tree) and an ***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.

During 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



(a) Parse tree (b) AST

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

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 descriptions assume the compiler is employing an AST representation of a syntax tree. See Figure 3.2 for a diagram of a typical compilation pipeline.

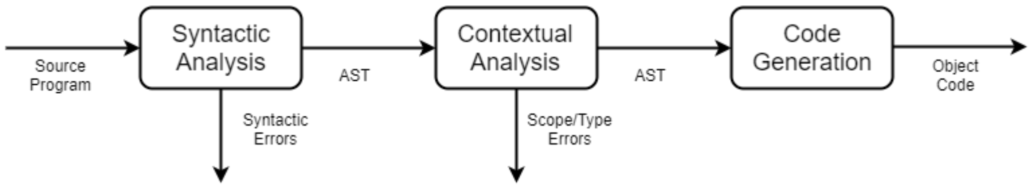
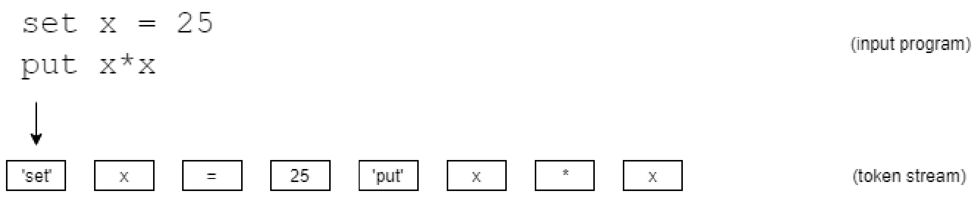


Figure 3.2: Typical compilation pipeline

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.



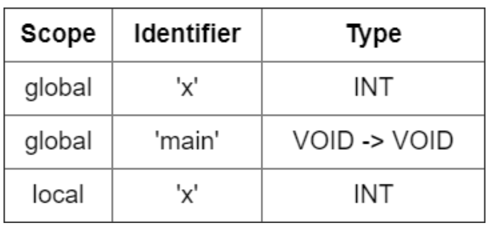
```
set x = 25
put x*x
```

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, which 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

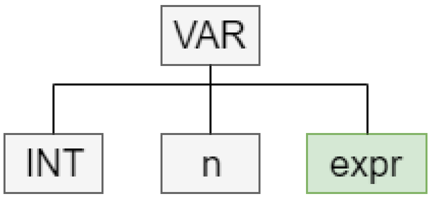
Upon successful completion of syntactic analysis, the generated AST is traversed by a contextual analyser. A contextual analyser checks whether the source program represented by the AST conforms to the source language's scope and type rules. Contextual analysis utilises an auxiliary data structure called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



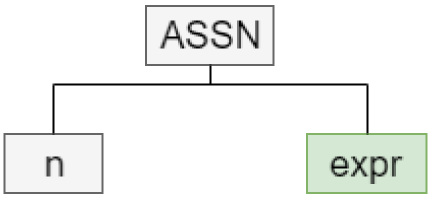
Scope	Identifier	Type
global	'x'	INT
global	'main'	VOID -> VOID
local	'x'	INT

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 look up 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



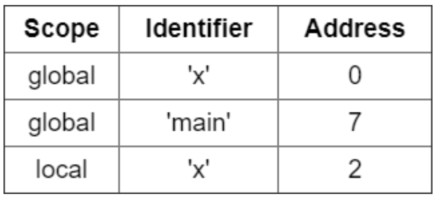
(b) Variable assignment

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

Type checking ensures that every operation in the program has operands of the expected type. The rules used by the contextual analyser 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.

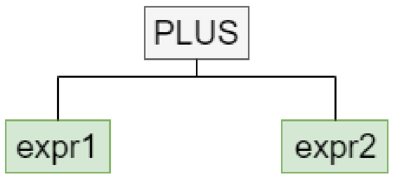
Code selection selects and generates the final object code. The developer of the compiler should plan which object code is selected for each construct encountered within the AST. The developer does this by devising a ***code template*** for each construct in the language.

3.4 Code Templates

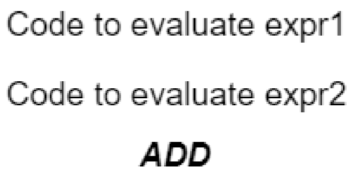
When developing a compiler, it is important to consider how each construct in the underlying language should be translated into object code. A code template provides a means of creating a theoretical model for each construct that specifies how the corresponding object code should be selected. Each template is simply 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, in Figure 3.7a 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 object code to perform their addition. 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.

In Figure 3.7b 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.



(a) AST



(b) Code template

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 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 discusses the requirements gathering phase of the application we intend to build, which we will henceforth refer to as the *FunCompiler*. Firstly, we detail the methods by which the requirements were collected. Secondly, we specify the FunCompiler's user stories, functional requirements and non-functional requirements. Finally, we attempt to summarise the functional requirements by creating a description of how an individual may use the application.

4.1 Methodology

The requirements of the FunCompiler were elicited through two main techniques. Firstly, through multiple interviews with Simon Gay, who is the current lecturer of the Programming Languages course at the University of Glasgow. Since the FunCompiler is being developed to act as an educational aid during this course, Simon clearly has the experience and expertise to offer valuable recommendations on the function and operation of the application. 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 formulated in Section 2.3. 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 step-wise, 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, we devise a set of user stories. User stories are short and simple descriptions of a feature, told from the perspective of a potential user:

- 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.

- As a user, I want to be able to input any Fun program, so that I can learn about the compilation process in the general case, not just for specific examples.
- As a user, I want to be able to view the animation of the contextual analysis phase of my program, so that I can understand how the compiler carries out this task.
- As a user, I want to be able to view the animation of the code generation phase of my program, so that I can understand how the compiler carries out this task.
- As a user, I want to be able to 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.
- 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.
- As a user, I want to be able to replay an animation, so that I can review any details I missed/misunderstood.
- As a user, I want to be able to step through the animation at my own pace, forwards or backwards, so I can more easily understand what is happening during the animation.

4.3 Functional Requirements

We then use the functionality as described within the user stories to create a formal list of functional requirements. Functional requirements are intended to capture a specific function of a system:

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 signifying the progress of the compiler's traversal.
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 any animations and report the syntax 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 contextual errors to the user).
16	Make the full specification of the Fun language available within the web application.

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).

4.4 Non-functional Requirements

Finally, we define a small list of 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 have a responsive design, at least for tablet-sized screens and larger.
4	The application must ensure no dangerous code can be executed, meaning no code that could potentially crash the client browser or the server machine.

4.5 Discussion

In an attempt to summarise the functional requirements we consider a potential workflow of the FunCompiler. Firstly, the user arrives at the website and should immediately be able to view the specification of the Fun language. Then, the user can type a Fun program into some form of code editor and choose to animate either contextual analysis or code generation. At this point, if there are any syntactic or contextual errors, they will be handled using the semantics as described in functional requirements **14** and **15**.

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 in functional requirements **10** to **13**) is simultaneously displayed elsewhere on screen. The user is of course free to pause the animation to review the current information visible, 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 workflow is sufficient to satisfy all functional requirements as defined above.

Chapter 5

Design

This chapter covers the aesthetic design process of the FunCompiler. We start by developing simple low-fidelity wireframe designs of the user interface's individual components and consider how each of these different components might satisfy a number of our functional requirements. We then look at a more complete design of the user interface and propose how a user might transition from page to page. Finally, we define how the application is to react upon encountering errors within the input Fun program. Note that full-size versions of the following designs (along with all other figures presented in this dissertation) can be found in Appendix D.

5.1 Application Interface

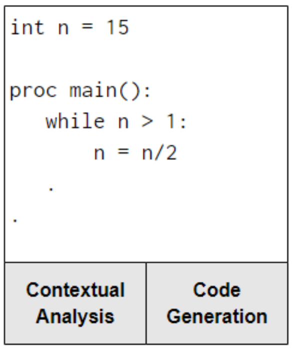
To begin realising the project's functional requirements we develop a series of quick and primitive wireframes, each representing an individual component of the user interface. For each component, we consider how it may meet one or more of the functional requirements (FRs) as defined in Chapter 4. Note that the application's non-functional requirements (NFRs) cover issues that are strictly more technical, and whilst they are not discussed explicitly, we do make indication of when each is satisfied throughout the remainder of this dissertation.

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 compilation of the input code and thus initiate the corresponding animation (FR **2**).

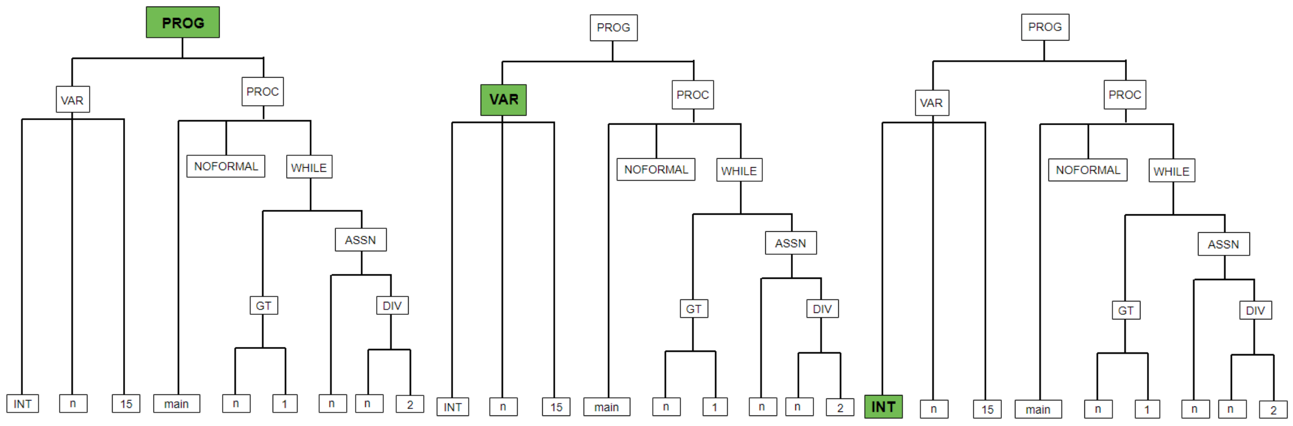
5.1.2 AST

The AST of the input program is the object over which the animation will occur. 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.



```
int n = 15

proc main():
   while n > 1:
       n = n/2
   .
.
```

Figure 5.1: Code editor wireframe



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

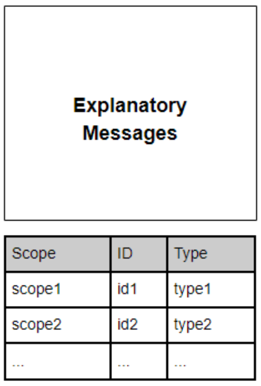
Figure 5.2: AST wireframes

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 items 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 designs.

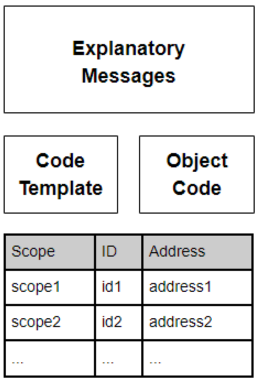
5.1.4 Playback Controls

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.



Scope	ID	Type
scope1	id1	type1
scope2	id2	type2
...	...	...

(a) Contextual analysis augmentations



Scope	ID	Address
scope1	id1	address1
scope2	id2	address2
...	...	...

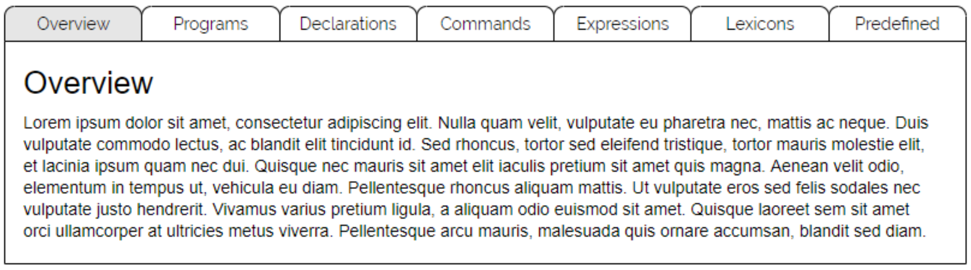
(b) Code generation augmentations

Figure 5.3: Augmentations wireframe

Figure 5.4: Playback controls wireframe

5.1.5 Fun Specification

Finally, as a matter a convenience, users may wish to have access to the full specification of the Fun language within the application. Figure 5.5, illustrates the specification broken down into seven sections, each of which covers a distinct concept of the language (FR **16**). Tabs along the top of the component allow the user to navigate between each section.



Overview

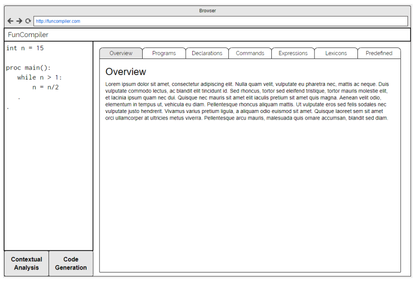
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla quam velit, vulputate eu pharetra nec, mattis ac neque. Duis vulputate commodo lectus, ac blandit elit tincidunt id. Sed rhoncus, tortor sed eleifend tristique, tortor mauris molestie elit, et lacinia ipsum quam nec dui. Quisque nec mauris sit amet elit iaculis pretium sit amet quis magna. Aenean velit odio, elementum in tempus ut, vehicula eu diam. Pellentesque rhoncus aliquam mattis. Ut vulputate eros sed felis sodales nec vulputate justo hendrerit. Vivamus varius pretium ligula, a aliquam odio euismod sit amet. Quisque laoreet sem sit amet orci ullamcorper at ultricies metus viverra. Pellentesque arcu mauris, malesuada quis ornare accumsan, blandit sed diam.

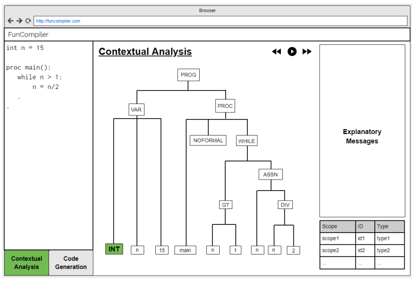
Figure 5.5: Fun specification wireframe

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 to navigate to a different page.

Figure 5.6 demonstrates the three main sections of the FunCompiler. Figure 5.6a demonstrates the landing page, in which the user has access to the code editor along with the full specification of the Fun language. Upon choosing to animate contextual analysis or code generation, the user is taken to the page shown in Figure 5.6b or 5.6c respectively. Here we see an AST generated from the input program, a set of playback controls for the animation and an area to display the relevant augmentations. Note that the user still has access to the code editor and is free to modify the input program and generate a new animation.


(a) Landing page wireframe


(b) Contextual analysis animation wireframe

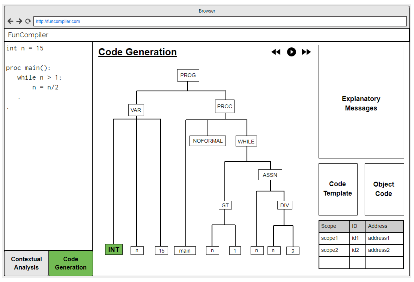
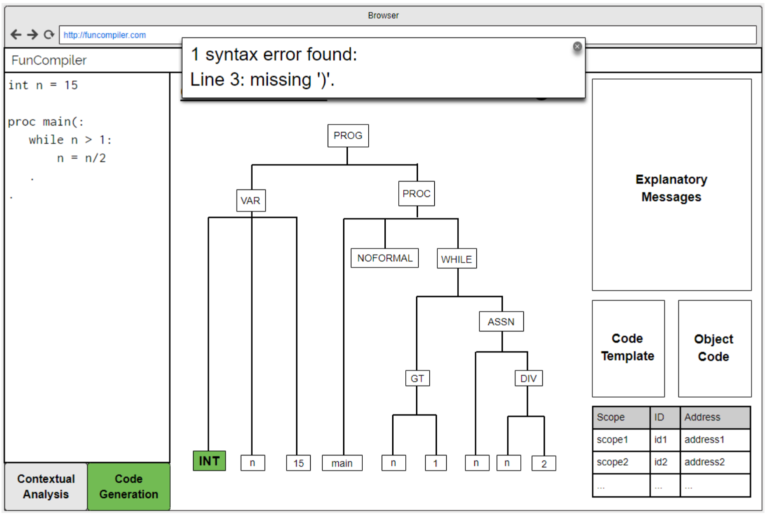

(c) Code generation animation wireframe

Figure 5.6: The three main sections of the FunCompiler

5.3 Error Handling

The above component designs successfully cover 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 that errors are found within the input Fun program. 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 and FR **15** states that if the input Fun program contains contextual errors, we may animate contextual analysis, but not code generation. In both cases, if the user tries to execute a prohibited action, they are presented with the relevant errors, as shown in Figure 5.7. In order to retain as much space as possible 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. With this final addition, all functional requirements of the FunCompiler have been successfully realised.


```
int n = 15

proc main(:
   while n > 1:
      n = n/2
      .
.
```
1 syntax error found:
Line 3: missing ')'.

Scope	ID	Address
scope1	id1	address1
scope2	id2	address2
...	...	...

Figure 5.7: 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.

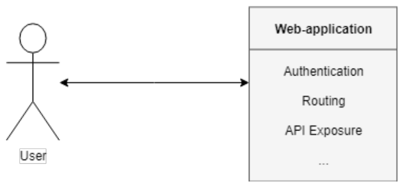
Firstly, we discuss the differences between a monolithic and a micro-service architecture in regards to web-applications. Secondly, we consider which of these architectures is most applicable to the FunCompiler and how the components of the system will be arranged to fit that model. Finally, using a 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

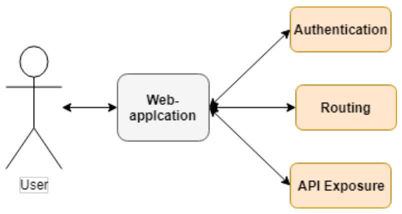
Traditionally, websites have been built as large, monolithic applications, where each unique service within the website (e.g., authentication, routing, API exposure) is encapsulated under one application, as illustrated in Figure 6.1a. Whilst a monolithic architecture may be an appropriate choice for 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* [10], 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 unfeasible.

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 [9]. 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 where any software validation provided by the original compiler no longer applies, and choice b) compels us to make a framework selection that is perhaps based more on a justification of convenience than it is prudence. Opting for choice c) means that the compiler need not 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 7.1.2).

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 (NFR **2**). Figure 6.2 illustrates this flow of information.

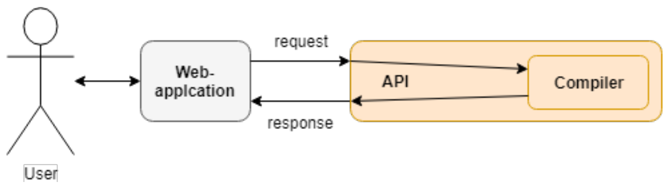


Figure 6.2: Web-application communicating with the compiler via an API

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-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 [2].

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 is 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 kinds 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 little setup and has extremely concise syntax, similar to that of functional languages, meaning we can create a REST API [17] with very few lines of code (as we will see in Section 7.1.1).

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.

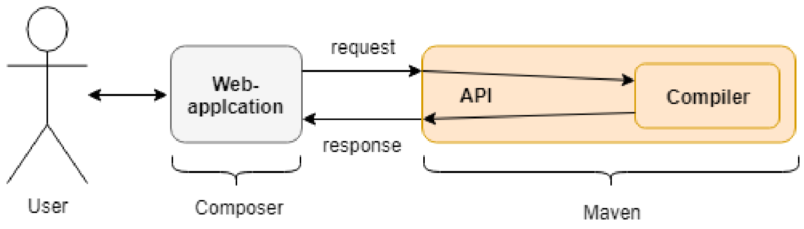


Figure 6.3: Current system infrastructure

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.

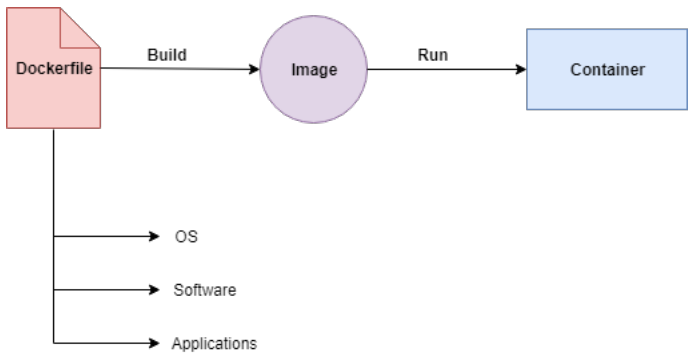


Figure 6.4: Creating a Docker container from an image

The FunCompiler contains two Dockerfiles which are used to build two different images: one for the web-application, one for the compiler. Whilst we will not 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 is 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, as long 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.

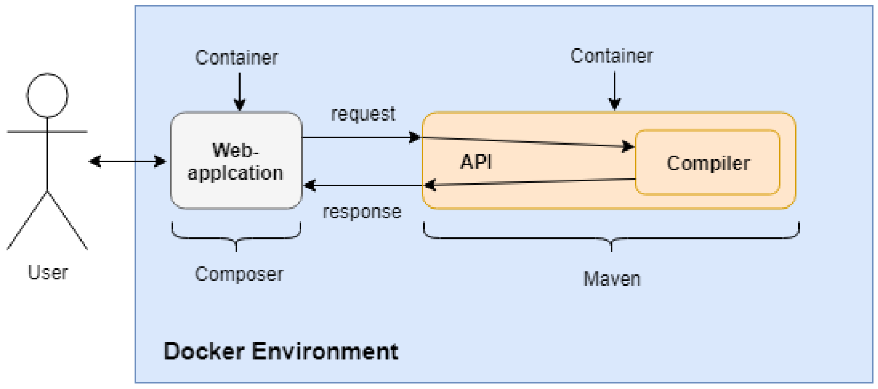


Figure 6.5: Final infrastructure of the FunCompiler

Chapter 7

Implementation

This chapter offers a high-level overview of the FunCompiler's implementation. As in Chapter 6, we decompose the system into two separate components: the web-application and the compiler. For each component, we examine the physical implementation details of any key aspects. We also briefly discuss the system's testing and continuous integration setup. Note that a full implementation of the FunCompiler can be accessed via haiti.dcs.gla.ac.uk:8000 (note that you must be connected to the University of Glasgow's School of Computing Science network).

7.1 Compiler Implementation

We begin by discussing how we convert the original, stand-alone compiler into the micro-service as described in Chapter 6.2. In order to accomplish this, two fundamental changes are necessary. Firstly, an API must be built on top of the compiler in order to enable communication between itself and the web-application. Secondly, the implementation of the compiler must be modified to return a large amount of structured data pertaining to the compilation of a particular program. With these changes in place, external services (such as the FunCompiler's web-application) are able to send API requests containing a Fun program to the compiler and in response receive an organised set of information that can be manipulated to build animations of the compilation process.

7.1.1 API Implementation

With reference to Section 6.3.2, recall that we intend to employ a Java micro-framework known as Spark to encapsulate the compiler and create a compact API with a single endpoint. If an external service desires to compile a Fun program, it must make a HTTP 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 and the value of `type` determines whether the compiler assembles information relevant to contextual analysis or code generation. Listing 7.1 contains the implementation of this endpoint in its entirety.

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

Listing 7.1: API implementation

To create the endpoint, we utilise Java 8 lambda expressions to define a POST route listening at the root URL. Within the body of the route we extract the values of the request's query parameters (`program` and `type`, as previously discussed) and supply them as arguments to the `execute()` method defined within the FunRun module, which is the driver method in regards to the compilation of a program. The return value of the `execute()` method is an object of the FunResponse class, of which each instance contains data collected during the compilation of a program. It is this data that we intend to return as a response to the calling entity, however, we must first utilise a Java library named GSON to serialise the contents of the FunResponse object into JSON, allowing it to be transmitted across a network.

7.1.2 Compiler Output & The FunResponse Class

To collect information relevant to the compilation of a program, we use a class named FunResponse. For each API request (and thus program to be compiled) received, a new object of this class is instantiated. Each 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).

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

Listing 7.2: FunResponse instance variables

In the interest of brevity, the details of the visitor/listener patterns used to derive the values of these variables are not discussed, however, we will analyse the typical contents of each variable and consider how the data contained within can be interpreted to build visualisations/animations.

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 do not assist in forming an animation. Their purpose is to provide a means of conveying to the calling service that errors were recognised within the supplied input program. For example, the web-application of the FunCompiler uses these values to report the errors to the user. In the case that the supplied input 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 can be interpreted to create and display a visual representation. 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.

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

Listing 7.3: Extract from a `treeNodes` array

Within this example there are three different types of node: `DIV`, `ID` and `NUM`. By examining the ID of the `DIV` node and the parent ID of the `ID` and `NUM` nodes, it is clear that `ID` and `NUM` are child nodes of `DIV`. By applying this knowledge, we can create a visual representation of the AST as derived from the above `treeNodes` array, as illustrated in Figure 7.1. We see that through inspection of the flat array returned from the API, external services can perceive the hierarchical relationships between the nodes of the AST, allowing the creation of visualisations.

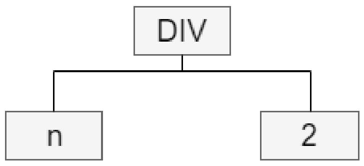


Figure 7.1: The visual representation of an AST as derived from Listing 7.3

Animation Order & Augmentations

After establishing a means of building a visual representation of the AST via the `treeNodes` array, information is still required that provides instruction in regards to specifically how the AST should be animated, i.e., in which order 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 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.

```
{
  "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"
    }
  ]
}
```

Listing 7.4: Extract from a `nodeOrder` array

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 relevance 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.

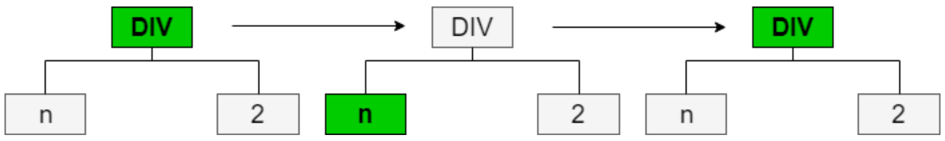


Figure 7.2: The animation of the AST as dictated by the Listings 7.3 and 7.4

We also see that we have fields for "explanations" and "table". These fields represent some of the augmentations that were discussed in Chapter 5. For each JSON object in nodeOrder, when the node represented by the ID field is highlighted during the animation, the value of fields such as "explanations" specify the type and value of the augmentations that should be associated with that particular step of the animation. Figure 7.3 demonstrates this concept by extending the animation shown in Figure 7.2 by inserting the "explanations" augmentation as specified in Listing 7.4. Note that 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.

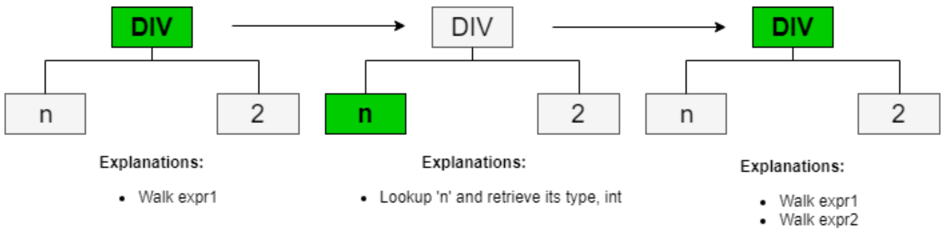


Figure 7.3: The animation of the AST as dictated by the Listings 7.3 and 7.4 with explanatory messages added

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 (e.g., explanatory messages, type table, etc.). This is extremely useful because, for example, we specified in Chapter 5 that the web-application is required to implement the playing, restarting and reversing of an animation. Since the animator can freely traverse forwards or backwards through the nodeOrder array whilst maintaining all the correct information, we can almost trivially implement these advanced playback controls, the specifics of which are discussed more in Section 7.2.4.

7.2 Web-application Implementation

In this section we describe the physical implementation of the interface designs we created in Chapter 5. For each component of the interface, we examine its role within the web-application, its aesthetic implementation and any interesting tools or libraries that were used to assist in its development. Note that whilst not discussed explicitly, the implementation of the FunCompiler's user interface does make extensive use of HTML, CSS, SASS, Bootstrap, vanilla Javascipt, jQuery and NodeJS.

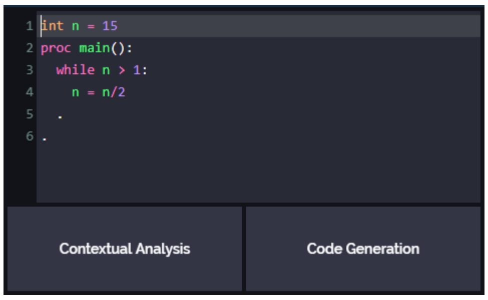
Whilst we do to some extent consider how the web-application interacts with the compiler API, once again in the interest of brevity, we will not discuss the minutiae of using Javascript to parse the contents of the response or to build the data structures that facilitate the animations.

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 browser [7]. CodeMirror provides syntax highlighting for over 100 predefined languages, bracket matching, auto-indentation and much more.

Unsurprisingly, Fun is not included as one of the languages that is provided with 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 features such as keywords, types and identifiers.

The implementation of the code editor is provided in Figure 7.4. We 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 the compiler, where 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.



```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

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 consecutively highlighting nodes to form an animation. We accomplish this using another Javascript library named D3 [8], which is a popular tool for data visualisation.

After sending an API request to the compiler by pressing one of the buttons in Figure 7.4, the web-application can utilise the information contained in the returned response to build a visualisation of the input program's AST. Similarly to how 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 (parent-child) representation of the AST.

D3 can now automatically parse this hierarchy to generate an SVG [14] representation of the AST. In Figure 7.5 we illustrate a series of ASTs generated from different Fun programs using D3. We see how, regardless of the number of the nodes within the AST (and thus the size of the input program), D3 intuitively shrinks or expands the layout of the tree to create the most eloquent visualisation possible.

The root node of each AST in Figure 7.5 demonstrates the aesthetics in regard to how the animator will highlight nodes during the traversal. Recall that to derive the order in which the nodes should be animated

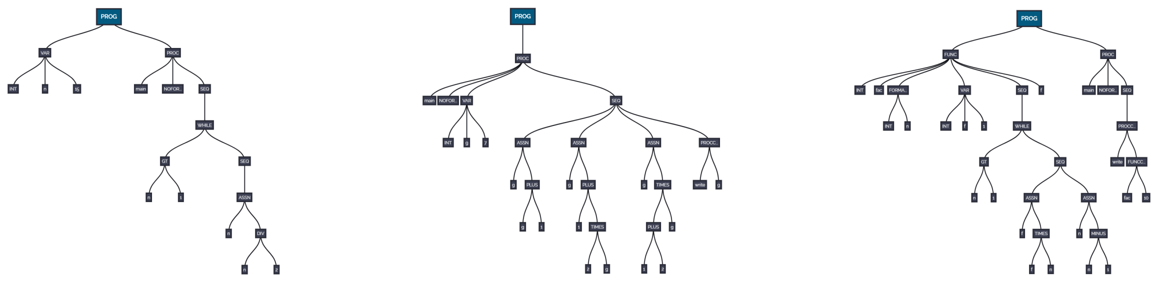
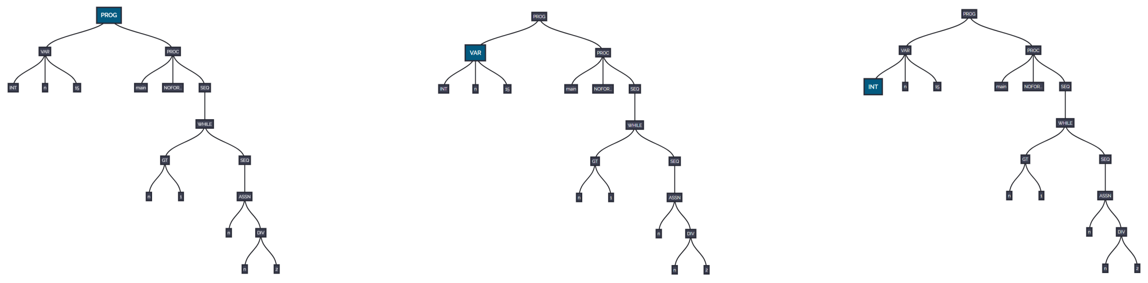


Figure 7.5: SVG representations of different Fun programs generated using D3

during the traversal, we need to iterate through the `nodeOrder` array which is also contained within the API response. Note that each node within the D3 generated SVG representation of the AST is itself an individual SVG component, which has an ID associated with it as specified in the `treeNodes` array. This means that when attempting to determine which node in the AST to highlight next, we can simply inspect the next ID specified in the `nodeOrder` array and highlight the SVG component with the corresponding ID. Figure 7.6 illustrates how nodes within the AST will be highlighted as the animation proceeds.



(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

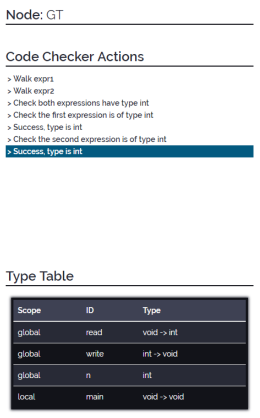
7.2.3 Augmentations Implementation

In Section 5.1.3 we created designs for the augmentations that would accompany each step of the animation. Simultaneously to using the ID field of each object in the `nodeOrder` array to consecutively highlight nodes within the AST, we can also retrieve the corresponding list of augmentations. The web-application can use the data contained within to build and display visual representations of the augmentations alongside the animation. Figure 7.7 illustrates the implementation of augmentations for both a contextual analysis animation and a code generation animation.

Note that for code template augmentations, we do not require the compiler to return the contents of each code template within the API response. Since the code template is fixed for each node type, we can simply examine the name of the node (as displayed at the top of each diagram in Figure 7.7) and load the corresponding code template from local storage.

7.2.4 Playback Controls Implementation

To implement the playback controls designed in Section 5.1.4 we simply utilise Bootstrap "glyphicons" as illustrated in Figure 7.8.



Node: GT

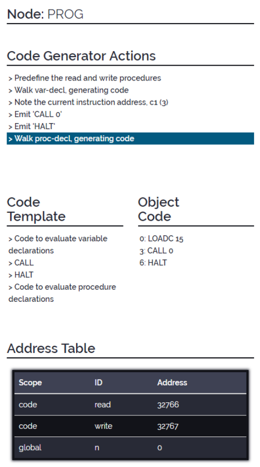
Code Checker Actions

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

Figure 7.8: Implementation of the playback controls

Each button has the following semantics with respect to the traversal of the `nodeOrder` array (recall by traversing through the array we are implicitly executing the animation over the AST):

- Pressing the play button will iterate through the array one element at a time at regular time intervals (e.g., every second) and replace the play button with a pause button.
- Pressing the pause button whilst the animation is playing will halt the iteration of the array at its current position and replace the pause button with a play button.
- Pressing the forwards/backwards button will move to the next or previous element in the array respectively. If the animation is playing when one of these buttons is pressed, it is automatically paused.
- Pressing the play button when the animation has finished will automatically restart the animation from the beginning.

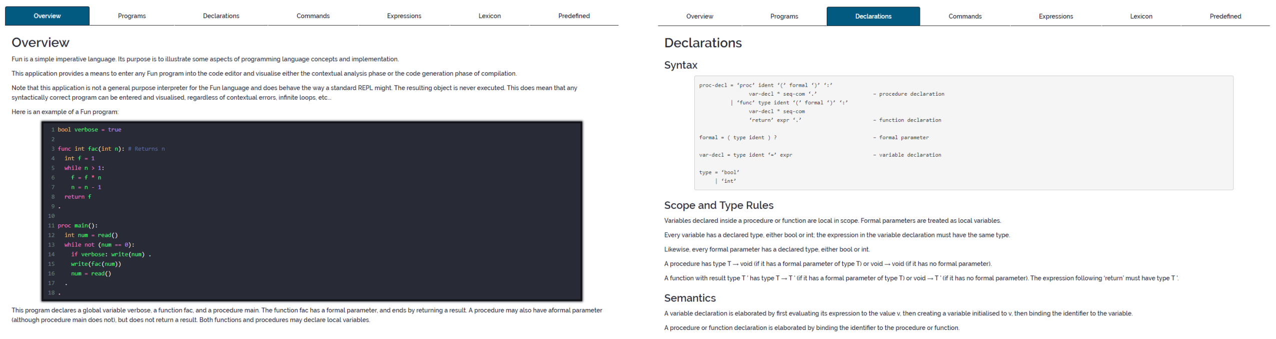
7.2.5 Fun Specification Implementation

The final interface component we are required to implement is the Fun specification as designed in Section 5.1.5. We embed the Fun specification into the web-application and create a tabbed navigation menu as illustrated in Figure 7.9. Each tab within the menu navigates to a different section of the specification.

7.2.6 Complete Implementation

Finally, we can combine each of these interface implementations to create the complete designs of the FunCompiler specified in Section 5.2. Figure 7.10 illustrates the full view of the landing page, the contextual analysis animation page and the code generation animation page.

Note that within the navigation menu we have added a "Home" button to return the user to the landing page



Overview

Fun is a simple imperative language. Its purpose is to illustrate some aspects of programming language concepts and implementation.

This application provides a means to enter any Fun program into the code editor and visualise either the contextual analysis phase or the code generation phase of compilation.

Note that this application is not a general purpose interpreter for the Fun language and does behave the way a standard REPL might. The resulting object is never executed. This does mean that any syntactically correct program can be entered and visualised, regardless of contextual errors, infinite loops, etc...

Here is an example of a Fun program:

```
bool verbose = true

func int fac(int n): # Returns 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()
  .
.
```

This program declares a global variable verbose, a function fac, and a procedure main. The function fac has a formal parameter, and ends by returning a result. A procedure may also have aformal parameter (although procedure main does not), but does not return a result. Both functions and procedures may declare local variables.

Declarations

Syntax

```
proc-decl = 'proc' ident '(' formal ')' ':'
                var-decl * seq-com '.'               - procedure declaration
          | 'func' type ident '(' formal ')' ':'
                var-decl * seq-com
                'return' expr '.'                    - function declaration

formal = ( type ident ) ?                            - formal parameter

var-decl = type ident '=' expr                       - variable declaration

type = 'bool'
     | 'int'
```

Scope and Type Rules

Variables declared inside a procedure or function are local in scope. Formal parameters are treated as local variables.

Every variable has a declared type, either bool or int; the expression in the variable declaration must have the same type.

Likewise, every formal parameter has a declared type, either bool or int.

A procedure has type T → void (if it has a formal parameter of type T) or void → void (if it has no formal parameter).

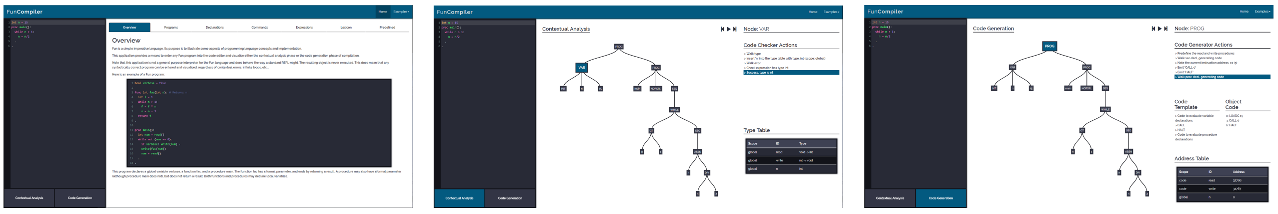
A function with result type T ' has type T → T ' (if it has a formal parameter of type T) or void → T ' (if it has no formal parameter). The expression following 'return' must have type T '.

Semantics

A variable declaration is elaborated by first evaluating its expression to the value v, then creating a variable initialised to v, then binding the identifier to the variable.

A procedure or function declaration is elaborated by binding the identifier to the procedure or function.

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 the FunCompiler

alongside a drop-drown menu of "Examples", illustrated in Figure 7.11. Each item in the drop-down menu corresponds to an example Fun program which when clicked, is automatically inserted into the code editor (these example programs are pulled from the "tests" directory of the Fun source code distributed to students at the University of Glasgow).

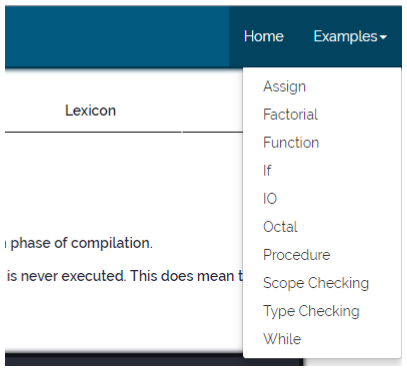


Figure 7.11: Drop-down menu of example Fun programs

7.3 Testing & Continuous Integration

Finally, we take a very brief look at the testing and continuous integration (CI) setup of the FunCompiler. For the purposes of this project, we treat the compiler (not including the API) as a third-party piece of software and assume that it has already been tested to an acceptable extent. Therefore, we focus our testing efforts on the correctness of the web-application and the API of the compiler.

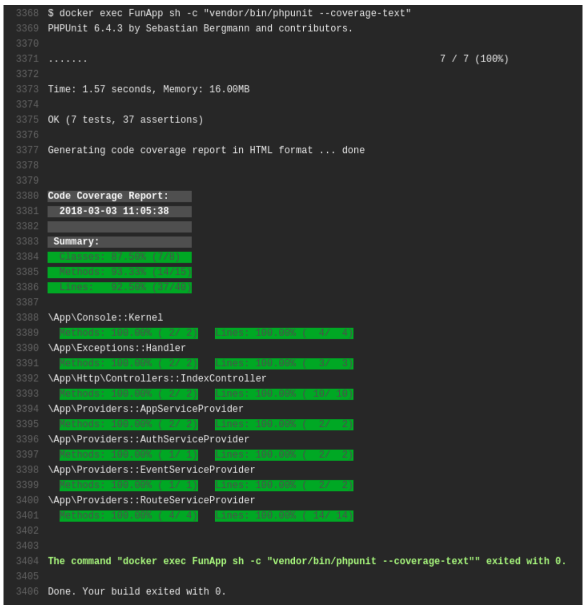
The test suite of the FunCompiler utilises PHPUnit [12], where the majority of tests are designed to ensure correct communication between the web-application and the compiler. In particular, we deliver API requests to the compiler and verify that the response received was as expected. For example, Listing 7.5 exhibits a test taken from the FunCompiler's test suite which sends an API request to the compiler which contains a syntactically invalid program (missing ':' at the end of the `while` statement).

```
public function testSuccessfulAndInvalidCARequest()
{
    $response = $this->json('POST', route('execute', [
        'type' => 'ca',
        'program' => 'int n = 15 proc main(): while n > 1 n = n/2 . .'
    ]));
    $response->assertSuccessful()
             ->assertJsonFragment([
                 'redirect_url' => '/',
                 'numSyntaxErrors' => 1,
                 'numContextualErrors' => 0,
                 'syntaxErrors' => ['line 1:36 missing \':\' at \'n\''],
             ])
             ->assertDontSeeText('nodeOrder')
             ->assertDontSeeText('treeNodes');
}
```

Listing 7.5: FunCompiler API test

The test looks to assert several factors, however, we are mainly seeking to verify that the API successfully included a syntax error with the correct error message, alongside not including the data structures used to create an animation (`treeNodes` and `nodeOrder`).

The FunCompiler is well integrated with Travis CI [6], meaning that each time we push changes to the FunCompiler's version control service (GitHub), Travis executes the entire build process (which, incidentally, is very straightforward due to the use of Docker) and then runs the test suite. This not only verifies that all tests pass successfully, but also detects any errors in the project's build. Figure 7.12 illustrates the FunCompiler's build completing successfully and all tests passing with a line coverage of 92.5%. Refer to Appendix A.5 to view the FunCompiler's Travis configuration file.



```
$ docker exec FunApp sh -c "vendor/bin/phpunit --coverage-text"
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.......                                                              7 / 7 (100%)

Time: 1.57 seconds, Memory: 16.00MB

OK (7 tests, 37 assertions)

Generating code coverage report in HTML format ... done


Code Coverage Report:
  2018-03-03 11:05:38

 Summary:
  Classes: 87.50% (7/8)
  Methods: 93.33% (14/15)
  Lines:   92.50% (37/40)

\App\Console::Kernel
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  4/  4)
\App\Exceptions::Handler
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  3/  3)
\App\Http\Controllers::IndexController
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% ( 10/ 10)
\App\Providers::AppServiceProvider
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  2/  2)
\App\Providers::AuthServiceProvider
  Methods: 100.00% ( 1/ 1)   Lines: 100.00% (  2/  2)
\App\Providers::EventServiceProvider
  Methods: 100.00% ( 1/ 1)   Lines: 100.00% (  2/  2)
\App\Providers::RouteServiceProvider
  Methods: 100.00% ( 4/ 4)   Lines: 100.00% ( 14/ 14)


The command "docker exec FunApp sh -c "vendor/bin/phpunit --coverage-text"" exited with 0.

Done. Your build exited with 0.
```

Figure 7.12: Successful build and test execution of the FunCompiler via Travis CI

Chapter 8

Evaluation

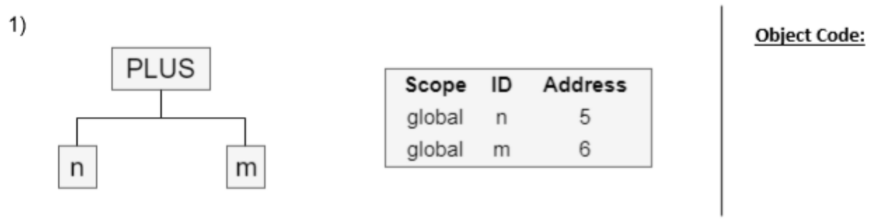
This chapter presents two separate evaluations, each aimed to assess a different aspect of the FunCompiler. The first evaluation considers the effectiveness of the application as an educational resource, while the second evaluation measures the quality of the application's user interface. For each evaluation, we offer the method by which the evaluation was conducted, the results that were attained and a discussion of the results implications.

8.1 Learning Evaluation

8.1.1 Method

One evaluation goal was to assess the effectiveness of the FunCompiler as a learning tool. We sought an evaluation method that would test if an individual's ability to learn or revise a concept was enhanced when using the system as an educational resource. To do this, a quiz was devised based on the concept of *code selection*, as described in Section 3.3.3. The quiz was attempted by ten participants, each of whom is a Glasgow University student currently enrolled in their fourth year of computer science. All participants successfully passed the course Programming Languages in their previous year of studies, meaning they had some prior understanding of the principles of code selection. The ten participants were divided into two groups of five, where one group was provided with access to the FunCompiler before attempting the quiz and the other was not. The participants using the FunCompiler will be referred to as the "animation" group and the other participants as the "non-animation" group.

The quiz consisted of ten questions. Each question was based on the notion of using an AST and an address table to predict which object code the compiler would generate after processing the AST. For example, Figure 8.1 exhibits a question taken from the quiz, where participants would be required to predict the object code of: *LOADG 5, LOADG 6, ADD*. Each participant's performance on the quiz was measured by a mark out of ten.



Scope	ID	Address
global	n	5
global	m	6

Figure 8.1: Example question taken from the evaluation quiz

Before the quiz, all participants were supplied with a reference sheet that provided an introduction to code selection using descriptions and diagrams. Those participants with access to the FunCompiler were encouraged to utilise the application as a means of better understanding the details found in the reference sheet. They were invited to try animating their own Fun programs or to use the "Examples" tab within the navigation menu to find a selection of predefined sample programs.

All participants were given a maximum of 15 minutes learning time prior to the start of the quiz, and 15 minutes to complete the quiz once they began. During the quiz, access to the FunCompiler was no longer allowed but all participants were permitted to retain access to the reference sheet. For more details on precisely how the evaluation was executed, please refer to Appendix B, which contains all evaluation materials including the reference sheet and the quiz.

8.1.2 Results

The results of the learning evaluation are presented in Table 8.1. The table illustrates each participant's score (out of ten) and calculates the average score of both groups. The results show that, on average, the animation group performed approximately 43% better than their non-animation counterparts.

	Score/10					
	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Average
Non-animation Group	6	4	5	4	9	**5.6**
Animation Group	10	7	8	8	7	**8**

Table 8.1: Learning evaluation results

8.1.3 Discussion

The purpose of the learning evaluation was two-fold. The primary objective was to demonstrate that the Fun-Compiler is an effective educational tool. That is, we wish to show that when attempting to learn particular aspects of compilation (such as code selection), the FunCompiler provides an advantage to simply learning from diagrams and text. The secondary objective was to provide supporting evidence for a theory we encountered in Chapter 2, the theory being that whilst algorithm animation might not be as helpful as we would intuitively believe, it is perhaps more beneficial to those looking to revise a concept, rather than learn it for the first time.

By the significant improvement in performance (43%) seen in the animation group compared to the non-animation group, it is reasonable to claim our primary objective has been satisfied. The improvement in test performance can likely be attributed to the fact that those with access to the FunCompiler were able to build the ASTs found in the reference sheet directly within the application. Consequently, when running the animation, participants were able to visualise exactly how the compiler would generate the object code for any given instance of the construct represented by the AST. Meanwhile, the participants in the non-animation group clearly struggled to apply the examples given within the reference sheet to the unseen problems in the quiz. That is, like discussed in Chapter 1, teaching abstract notions using predefined examples does not necessarily promote a generalised understanding of a topic.

Achieving the secondary objective of the evaluation proved to be much more difficult. The initial plan was to conduct the same evaluation as described above, but also include a second group of participants who were computer science students and had *not* studied the course Programming Languages before. The evaluation would have aimed to show that whilst the participants who had access to the FunCompiler performed better in both cases, the improvement in performance of those who were revising the material would be greater than in those learning it for the first time. Unfortunately, under the circumstances, gaining enough suitable participants to provide meaningful results was not possible. Therefore, this aspect of the evaluation was abandoned. Regardless, unlike the results of the studies discussed in Chapter 2, this evaluation did reveal that the animation was of significant benefit. Whilst, without a benchmark, we cannot say that this is definitively due to the fact that the participants were revising, it is perhaps an interesting consideration to keep in mind.

8.2 User Interface Evaluation

8.2.1 Method

The second evaluation goal was to assess the quality of the FunCompiler's user interface. After the participants had finished the quiz, those that had access to the FunCompiler (five participants) were asked to complete a short questionnaire aimed to evaluate their experience with the user interface. The questionnaire is heavily based on the *Questionnaire for User Interface Satisfaction* (QUIS) which is a standardised and popular method of measuring the quality of a user interface, originally designed at the University of Maryland [5].

Again, the full questionnaire can be found in Appendix B, but in short, the questionnaire is split into five categories, each consisting of questions presenting the participant with a typical characteristic of a user interface and asking them to rate their experience with that feature on a scale of 0-9, where each scale is provided with a specific interpretation. For example, Figure 8.2 illustrates one of the questions taken from the questionnaire, which asks the user to rate their experience in regards to the positioning of messages, 0 being very inconsistent, 9 being very consistent.

13. Position of messages on screen inconsistent consistent

Figure 8.2: Example question taken from the questionnaire

8.2.2 Results

Whilst we won't discuss the contents of each response individually, we do collect all the responses gathered for each category and provide the average score on a scale of 0 to 9. A high score indicates a positive reaction to that particular category and a low score indicates a negative reaction.

- Screen: concerns the readability and organisation of information on screen. **Average score: 8.25**.
- Terminology and System Information: concerns the consistent use of terms and appropriate presentation of feedback. **Average score: 7**.
- Learning: concerns the ease with which users can understand how to operate the system. **Average score: 7.7**.
- System Capabilities: concerns the high-level factors of the system including speed and reliability. **Average Score: 8.3**.
- Overall Reaction to the Software: concerns general aspects including flexibility and ease of use. **Average score: 8.1**.

8.2.3 Discussion

In general, the feedback to the user interface evaluation was overwhelmingly positive. The majority of participants felt it was fast, intuitive and aesthetically pleasing. The issues the participants did have mainly revolved around the concept of learning how to initially use the application along with which features were available. Many participants felt that, whilst the embedded specification was useful for recapping the semantics of the Fun language, it did not necessarily contain any information explaining what the application did or how it worked. Overall, the participants felt that access to a user guide would certainly be of great benefit. It was initially considered unnecessary to expand the size of the application to embed a user guide given that, in most cases, students would be directed to this application by a lecturer, who would demonstrate its usage in class. However, a resolution to this issue would be to host a user guide externally (as a PDF download, perhaps) that could be accessed via a link within the FunCompiler. This way, the size of the application need not be expanded (other than to include a single link), and all users would have access to materials detailing the application's operation. Consequently, a user guide was created and is included in Appendix C.

Whilst the FunCompiler does have a robust responsive design (NFR **3**), some participants did point out that the layout which certain visual elements conform to on smaller screens makes the animation slightly difficult to interpret due to the fact that one cannot see the AST and the augmentations within the same view. Unfortunately, this comes as a consequence of an application that demands a large amount of visual information to be displayed simultaneously in the same perspective. Currently, it is not immediately clear how this problem could be resolved but a redesign of the FunCompiler's responsive system could be listed for future work. Whilst using the application from a mobile phone is certainly still possible, for the time being, users are advised to use tablet or larger sized screens for the best experience.

We take this opportunity to note that if this evaluation had been repeated, we could have taken advantage of the fact that QUIS is a standardised technique. Meaning, we could have asked the same participants to complete the questionnaire for a similar piece of algorithm animation software, such as VisuAlgo discussed in Chapter 2. Being that VisuAlgo is one of the most well-implemented algorithm animation applications currently available, the results of this questionnaire would have provided an appropriate benchmark to compare against the FunCompiler. This comparison could have been an effective tool for guiding which areas of the application required the most work.

Chapter 9

Conclusion

9.1 Summary

The aim of this project was to develop a web-application that could be used as an educational resource when attempting to learn or teach certain concepts of compilation. The result of this project allows users to express the abstract notions characterised by compilation as concrete visualisations. The web-application provides the ability to generate a dynamic animation of the compilation process, which traverses over the AST built from a Fun program of the user's choice. The user can choose to animate either the contextual analysis or code generation phase of compilation, where each animation is augmented with supplementary information to further assist the user's understanding.

The final implementation of the FunCompiler comfortably achieves all the requirements as collected in Chapter 4 along with completing several extra features. The evaluation conducted in Chapter 8 presents suitable evidence to show that the system can be utilised as an effective learning tool whilst simultaneously providing an enjoyable user experience. The application has already been deployed by the University of Glasgow's Computing Science department at haiti.dcs.gla.ac.uk:8000 and will hopefully be integrated into future instalments of the Programming Languages course, providing a benefit to both the students and the lecturer. The system has been tested and known to be compatible with (at least) the most recent versions of Google Chrome, Firefox, Microsoft Edge and Internet Explorer (NFR **1**).

9.2 Lessons Learned

Throughout the course of this project, the author has further solidified his understanding of web-application development in general, but also, through working extensively with ANTLR and implementing custom listeners/visitors, has gained a much deeper insight into the finer details of compiler operation and behaviour. Additionally, the author has witnessed first-hand the benefits of a well-implemented system architecture, whereby, through the use of tools like the Docker, the development and deployment of a diverse project such as this one became effortless.

Primarily, the author has developed a new found respect for the importance of efficient evaluation planning. That is, to have learned that without the correct participants and the appropriate environment, a well-conceived evaluation goal alone is not sufficient. Had the evaluation been carried out more effectively, we would have been able to use the FunCompiler as the ideal tool to more closely investigate the claims and theories we encountered in our background research of algorithm animation.

9.3 Future Work

9.3.1 Interpretation of Object Code

During the evaluation, more than one participant questioned the author as to the whereabouts of the program's output, i.e., the value of any `write()` statements. Given the nature of the application, these participants seemed to expect that the generated object code would be executed (or, realistically, interpreted). Interpretation of the object code was a feature that, whilst certainly possible, was both not within the scope of the project's specification and considered to have several drawbacks which resulted in its inclusion being rejected. These drawbacks include handling indefinitely large outputs and dealing with programs that contain harmful runtime issues. These runtime issues include problems such as infinite loops, which could cause either the client's browser or the server machine to crash, which would require complex mechanisms to efficiently prevent.

An advantage of removing interpretation is that any program, even one with infinite loops, can be submitted and animated in the FunCompiler without causing any issues for either the client or the server (NFR **4**). However, it is the case that, with a large amount of extra development time, we could retain the same functionality but allow the execution of object code by implementing the appropriate safety mechanisms (for example, timeouts). Execution of the object code would allow the FunCompiler to behave as an interactive shell (still including all the existing functionality), comparable to modern REPLs [13].

9.3.2 Inclusion of More Languages

In Section 3.1 we discussed how the Fun language was general enough that the concepts learned using Fun could easily be applied to other languages. Regardless, some users, especially those who are not studying the Programming Languages course, may prefer that the application was based on a language such as C, Python, or any other language of their choosing.

An especially useful feature would be the ability for users to select the language they were looking to write/animate their programs in. Given the project's powerful setup, if supplied with the appropriate ANTLR files to support each language (a grammar and any relevant visitors/listeners), this feature could be enabled with relative ease. This would create an application that was more accessible to a far wider audience.

9.4 Acknowledgements

I would like to take this opportunity to thank Simon Gay for his assistance in the initial requirements gathering phase and Ornela Dardha for her continual guidance and enthusiastic involvement throughout the course of this project.

Appendices

Appendix A

Code Listings

A.1 Maven Project Object Model

```
<?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>

  <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>
```

Listing A.1: pox.xml

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/
   bin --filename=composer
# 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
```

Listing A.2: Dockerfile

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
   container
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"]
```

Listing A.3: Dockerfile

A.4 Docker Compose File

```
# 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
```

Listing A.4: docker-compose.production.yml

A.5 Travis Configuration File

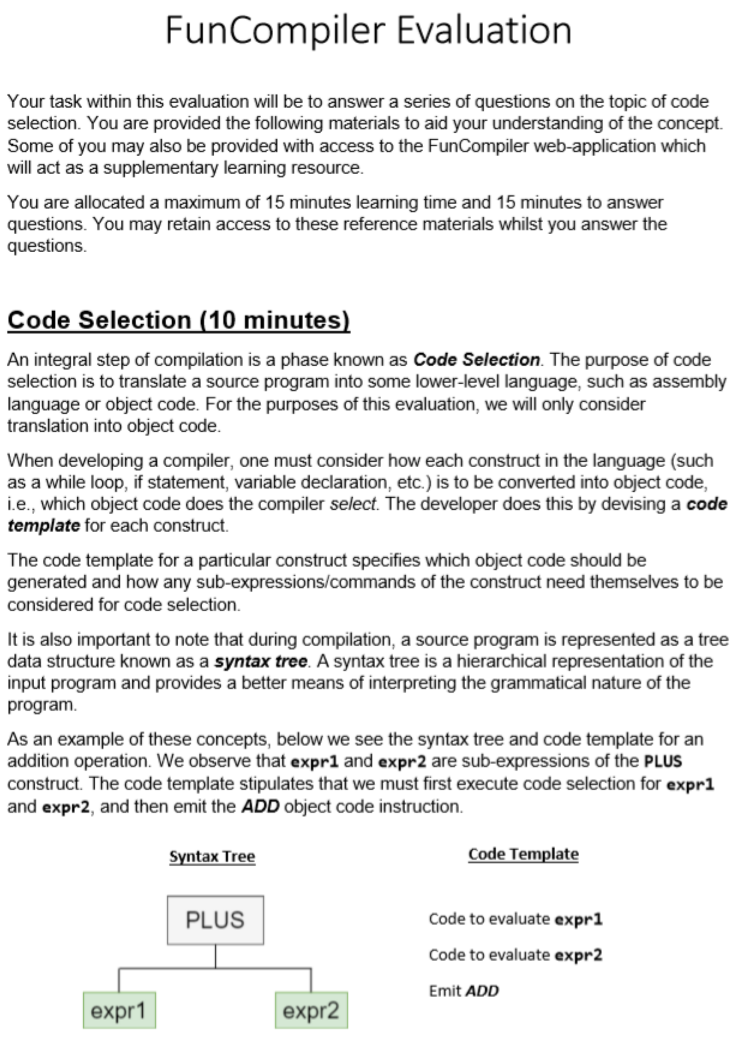
```
sudo: required

services:
  - docker

script:
  - docker-compose -f docker-compose.production.yml up -d
  - docker exec FunApp sh -c "vendor/bin/phpunit --coverage-text"
```

Listing A.5: travis.yml

Appendix B

Evaluation Materials

B.1 Quiz Reference Materials



FunCompiler Evaluation

Your task within this evaluation will be to answer a series of questions on the topic of code selection. You are provided the following materials to aid your understanding of the concept. Some of you may also be provided with access to the FunCompiler web-application which will act as a supplementary learning resource.

You are allocated a maximum of 15 minutes learning time and 15 minutes to answer questions. You may retain access to these reference materials whilst you answer the questions.

Code Selection (10 minutes)

An integral step of compilation is a phase known as ***Code Selection***. The purpose of code selection is to translate a source program into some lower-level language, such as assembly language or object code. For the purposes of this evaluation, we will only consider translation into object code.

When developing a compiler, one must consider how each construct in the language (such as a while loop, if statement, variable declaration, etc.) is to be converted into object code, i.e., which object code does the compiler *select*. The developer does this by devising a ***code template*** for each construct.

The code template for a particular construct specifies which object code should be generated and how any sub-expressions/commands of the construct need themselves to be considered for code selection.

It is also important to note that during compilation, a source program is represented as a tree data structure known as a ***syntax tree***. A syntax tree is a hierarchical representation of the input program and provides a better means of interpreting the grammatical nature of the program.

As an example of these concepts, below we see the syntax tree and code template for an addition operation. We observe that **expr1** and **expr2** are sub-expressions of the **PLUS** construct. The code template stipulates that we must first execute code selection for **expr1** and **expr2**, and then emit the ***ADD*** object code instruction.

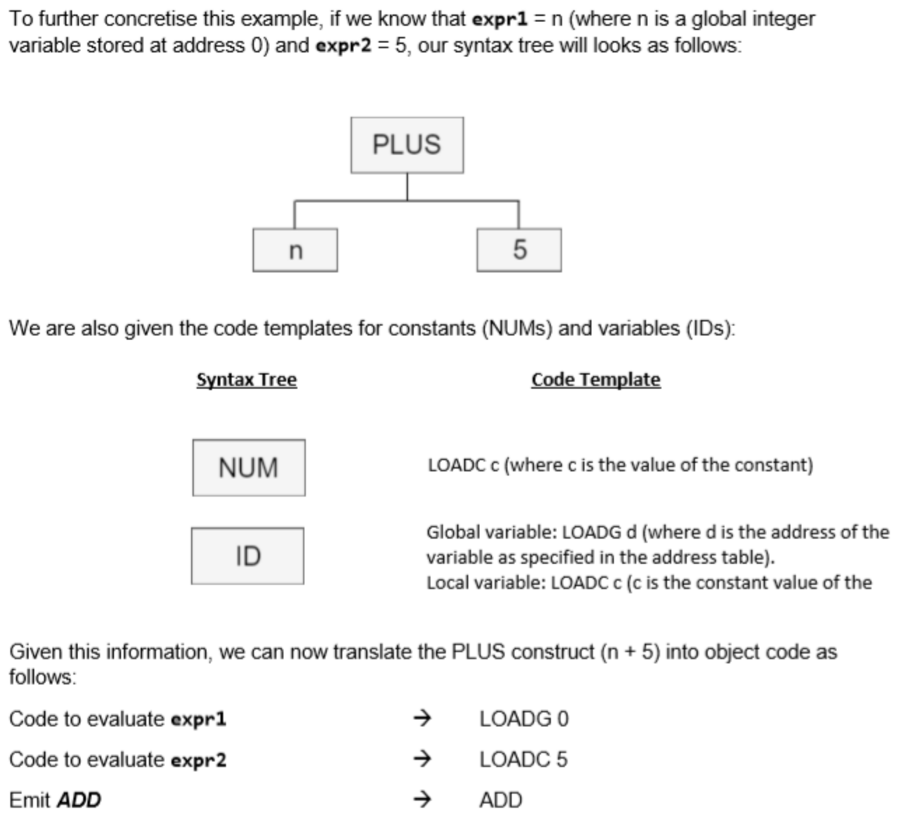
Code Template

Code to evaluate **expr1**

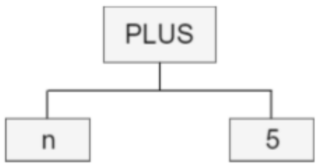
Code to evaluate **expr2**

Emit ***ADD***

Figure B.1: Quiz reference materials 1



To further concretise this example, if we know that **expr1** = n (where n is a global integer variable stored at address 0) and **expr2** = 5, our syntax tree will looks as follows:



We are also given the code templates for constants (NUMs) and variables (IDs):

Syntax Tree	Code Template
NUM	LOADC c (where c is the value of the constant)
ID	Global variable: LOADG d (where d is the address of the variable as specified in the address table). Local variable: LOADC c (c is the constant value of the

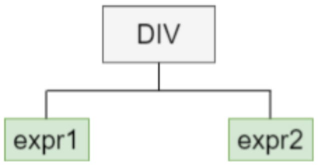
Given this information, we can now translate the PLUS construct (n + 5) into object code as follows:

Code to evaluate **expr1**	→	LOADG 0
Code to evaluate **expr2**	→	LOADC 5
Emit ***ADD***	→	ADD

Figure B.2: Quiz reference materials 2

This page acts as a resource sheet, containing the syntax tree and code template for the various constructs you will see in the following questions.

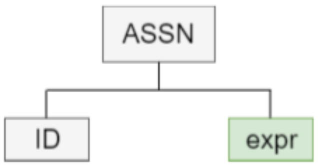
Syntax Tree | **Code Template**



Code to evaluate **expr1**

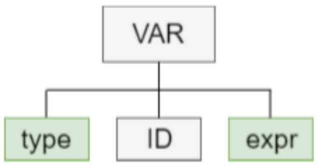
Code to evaluate **expr2**

Emit ***DIV***

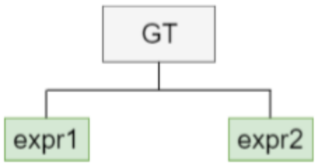


Code to evaluate **expr**

STOREG d or ***STOREL*** d (where d is the offset of the global or local variable)



Code to evaluate **expr**



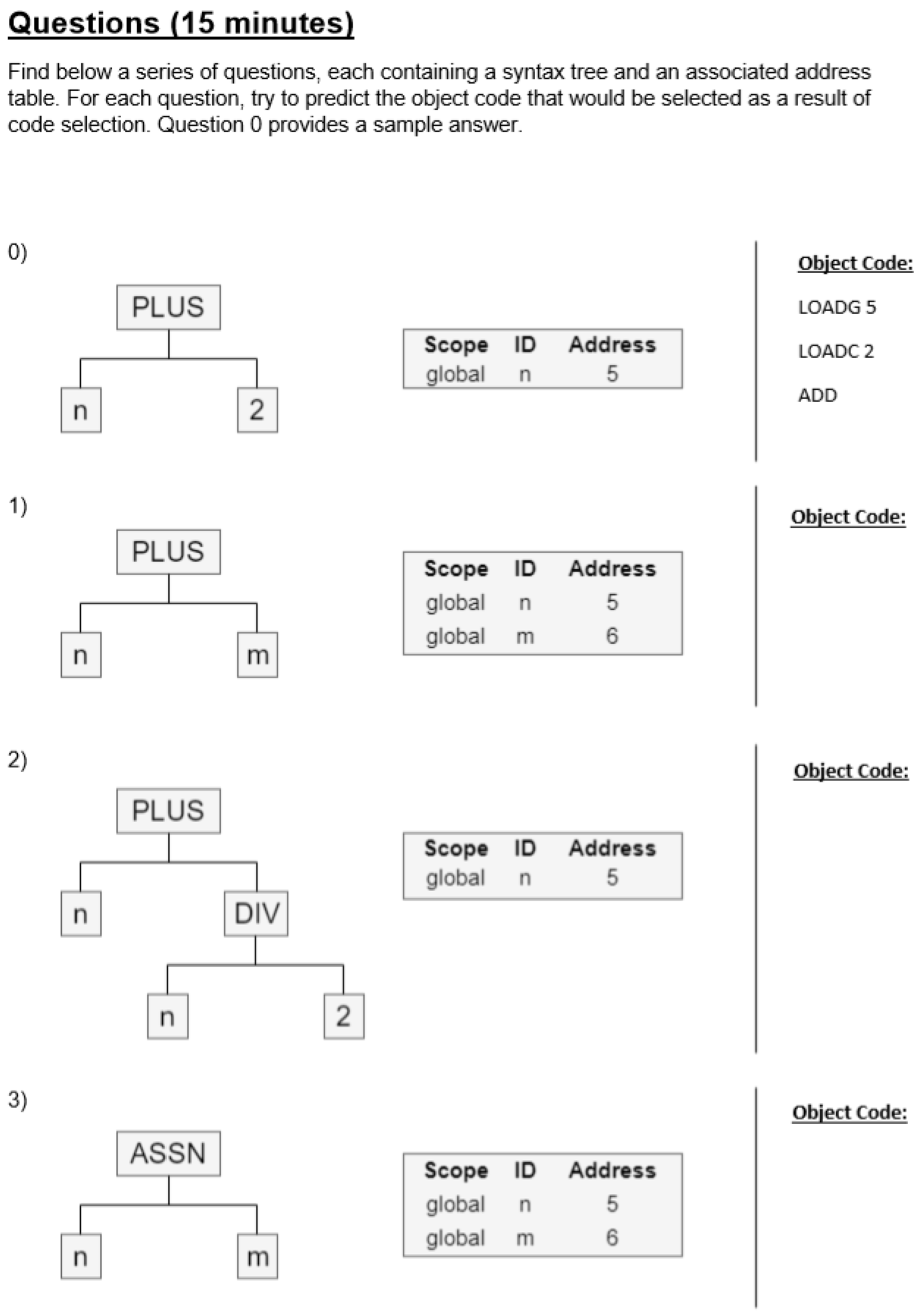
Code to evaluate **expr1**

Code to evaluate **expr2**

Emit ***GT***

Figure B.3: Quiz reference materials 3

B.2 Quiz



Questions (15 minutes)

Find below a series of questions, each containing a syntax tree and an associated address table. For each question, try to predict the object code that would be selected as a result of code selection. Question 0 provides a sample answer.

0)

PLUS
n
2

Scope	ID	Address
global	n	5

Object Code:

LOADG 5

LOADC 2

ADD

1)

PLUS
n
m

Scope	ID	Address
global	n	5
global	m	6

Object Code:

2)

PLUS
n
DIV
n
2

Scope	ID	Address
global	n	5

Object Code:

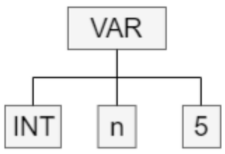
3)

Scope	ID	Address
global	n	5
global	m	6

Object Code:

Figure B.4: Quiz 1

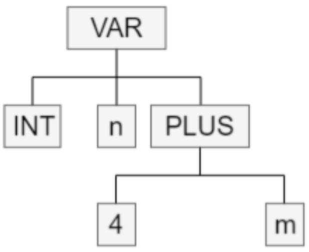
4)



Scope	ID	Address
global	n	5

Object Code:

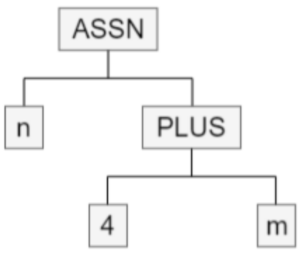
5)



Scope	ID	Address
global	n	5
global	m	6

Object Code:

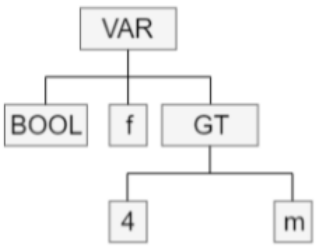
6)



Scope	ID	Address
global	n	5
global	m	6

Object Code:

7)

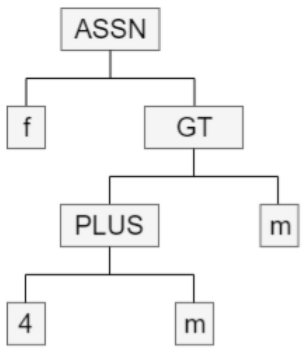


Scope	ID	Address
global	f	7
global	m	6

Object Code:

Figure B.5: Quiz 2

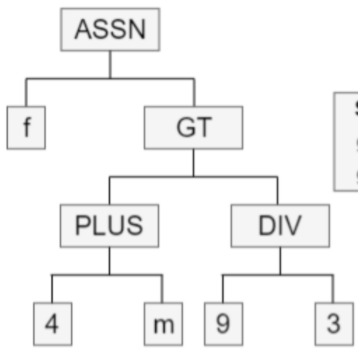
8)



Scope	ID	Address
global	f	7
global	m	6

Object Code:

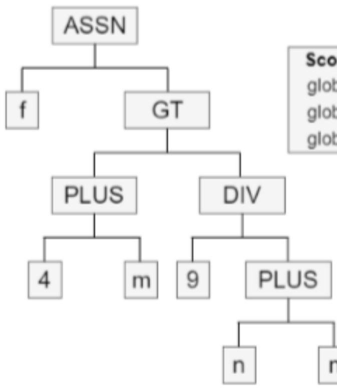
9)



Scope	ID	Address
global	f	7
global	m	6

Object Code:

10)

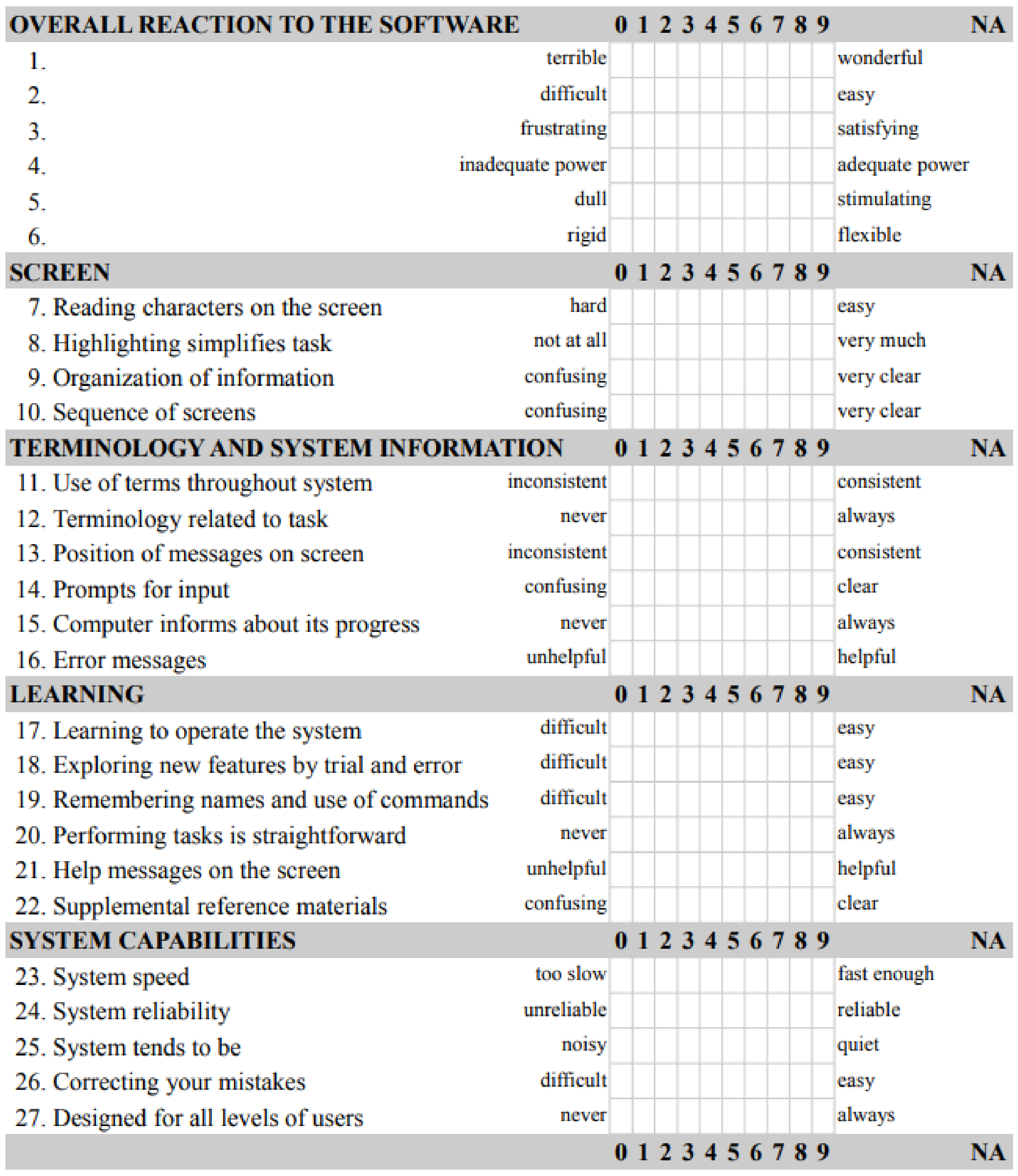


Scope	ID	Address
global	f	7
global	m	6
global	n	5

Object Code:

Figure B.6: Quiz 3

B.3 Questionnaire

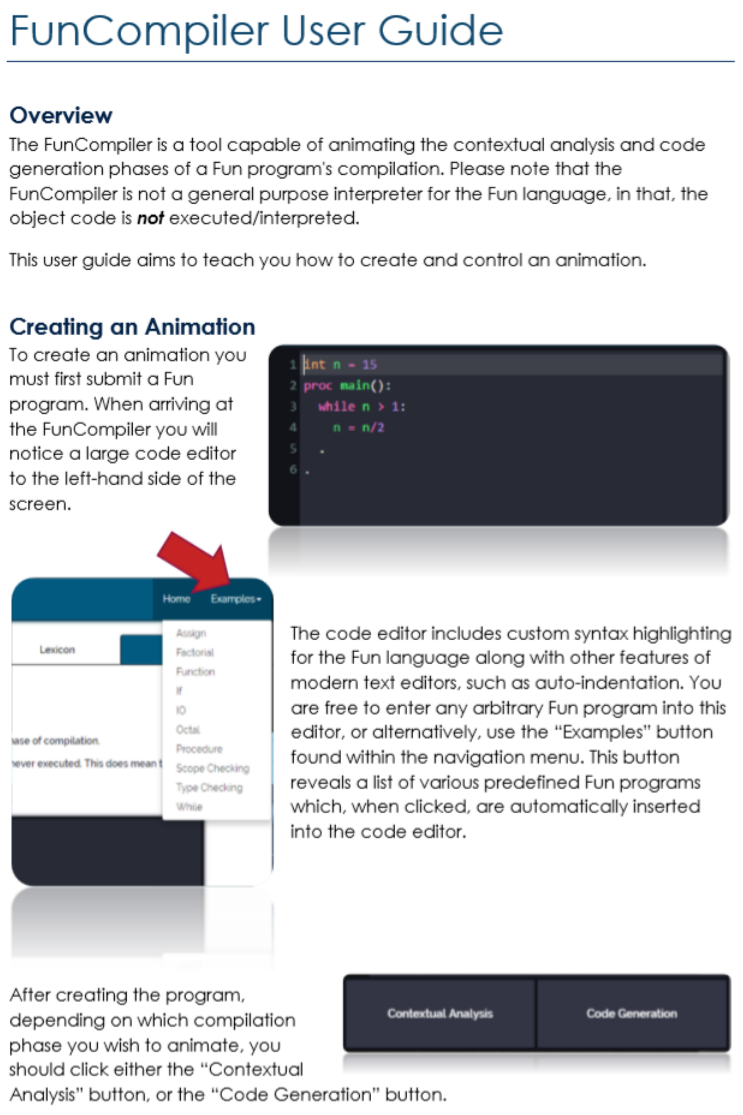


OVERALL REACTION TO THE SOFTWARE		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**NA**
1.	terrible											wonderful	
2.	difficult											easy	
3.	frustrating											satisfying	
4.	inadequate power											adequate power	
5.	dull											stimulating	
6.	rigid											flexible	
SCREEN		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**NA**
7. Reading characters on the screen	hard											easy	
8. Highlighting simplifies task	not at all											very much	
9. Organization of information	confusing											very clear	
10. Sequence of screens	confusing											very clear	
TERMINOLOGY AND SYSTEM INFORMATION		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**NA**
11. Use of terms throughout system	inconsistent											consistent	
12. Terminology related to task	never											always	
13. Position of messages on screen	inconsistent											consistent	
14. Prompts for input	confusing											clear	
15. Computer informs about its progress	never											always	
16. Error messages	unhelpful											helpful	
LEARNING		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**NA**
17. Learning to operate the system	difficult											easy	
18. Exploring new features by trial and error	difficult											easy	
19. Remembering names and use of commands	difficult											easy	
20. Performing tasks is straightforward	never											always	
21. Help messages on the screen	unhelpful											helpful	
22. Supplemental reference materials	confusing											clear	
SYSTEM CAPABILITIES		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**NA**
23. System speed	too slow											fast enough	
24. System reliability	unreliable											reliable	
25. System tends to be	noisy											quiet	
26. Correcting your mistakes	difficult											easy	
27. Designed for all levels of users	never											always	
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**NA**

Figure B.7: Questionnaire

Appendix C

User Guide



FunCompiler User Guide

Overview

The FunCompiler is a tool capable of animating the contextual analysis and code generation phases of a Fun program's compilation. Please note that the FunCompiler is not a general purpose interpreter for the Fun language, in that, the object code is ***not*** executed/interpreted.

This user guide aims to teach you how to create and control an animation.

Creating an Animation

To create an animation you must first submit a Fun program. When arriving at the FunCompiler you will notice a large code editor to the left-hand side of the screen.

```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

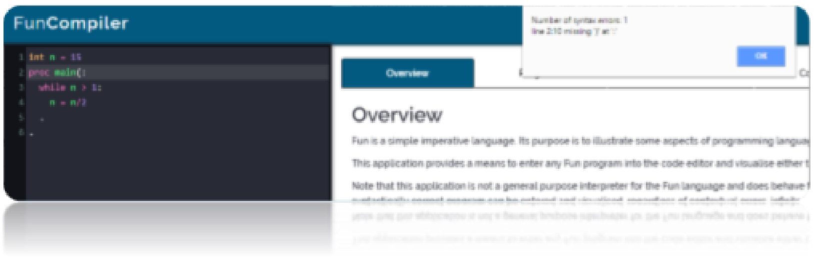
ase of compilation.
never executed. This does mean t

The code editor includes custom syntax highlighting for the Fun language along with other features of modern text editors, such as auto-indentation. You are free to enter any arbitrary Fun program into this editor, or alternatively, use the "Examples" button found within the navigation menu. This button reveals a list of various predefined Fun programs which, when clicked, are automatically inserted into the code editor.

After creating the program, depending on which compilation phase you wish to animate, you should click either the "Contextual Analysis" button, or the "Code Generation" button.

Figure C.1: User guide page 1

Note that there are of course, some limitations on what code can be submitted. Whether animating contextual analysis or code generation, the program must be syntactically valid. If syntax errors are present, they will be reported when attempting to create the animation. On the other hand, the program must be contextually valid only when animating code generation. Meaning, if the program contains contextual errors, animation of contextual analysis is still permitted (and will in fact reveal the cause of the errors!)



```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

Number of syntax errors: 1
line 2:10 missing ')' at ':'

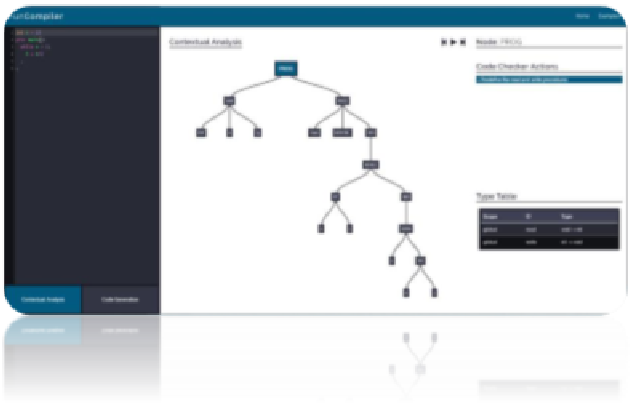
Overview

Fun is a simple imperative language. Its purpose is to illustrate some aspects of programming langua

This application provides a means to enter any Fun program into the code editor and visualise either t

Note that this application is not a general purpose interpreter for the Fun language and does behave t

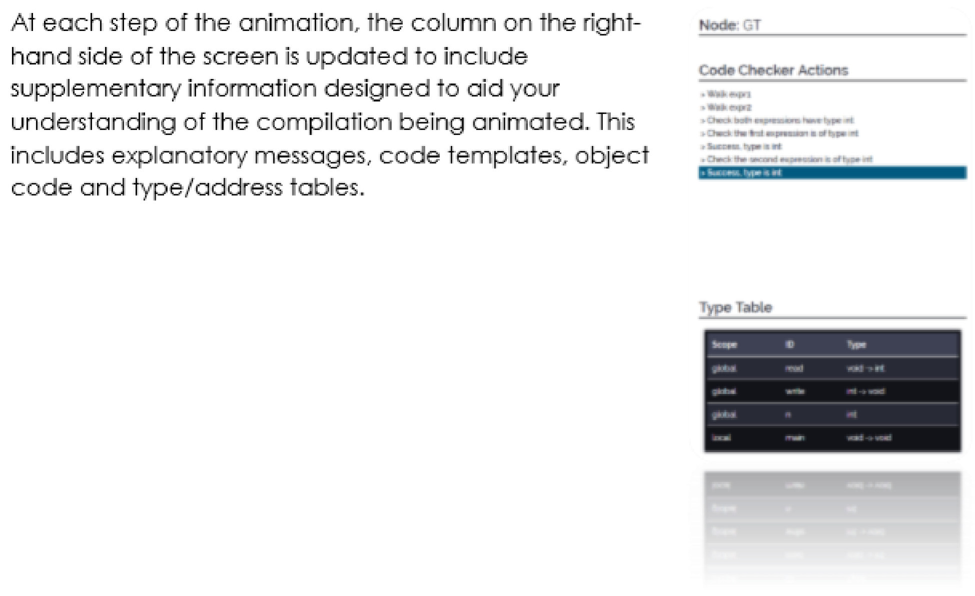
Controlling an Animation



After choosing to animate one of the compilation phases, you will be presented with the abstract syntax tree generated from your input program.

Above the animation there are a set of playback buttons, allowing you to play, pause, or step through the animation (forwards and backwards).

Figure C.2: User guide page 2



At each step of the animation, the column on the right-hand side of the screen is updated to include supplementary information designed to aid your understanding of the compilation being animated. This includes explanatory messages, code templates, object code and type/address tables.

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

Figure C.3: User guide page 3

Appendix D

Full-size Images

Note: if you are viewing this dissertation via a PDF, all images within the appendix are linked to their corresponding versions in the body, and vice versa.

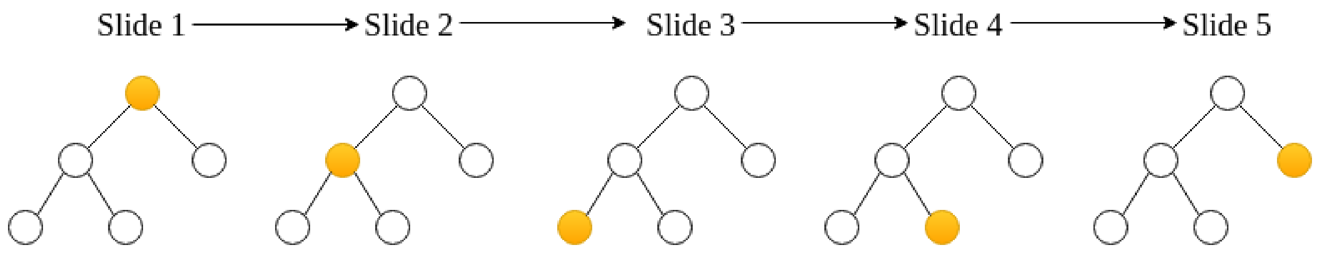


Figure 1.1: A sequence of slides demonstrating a depth-first traversal over a binary tree

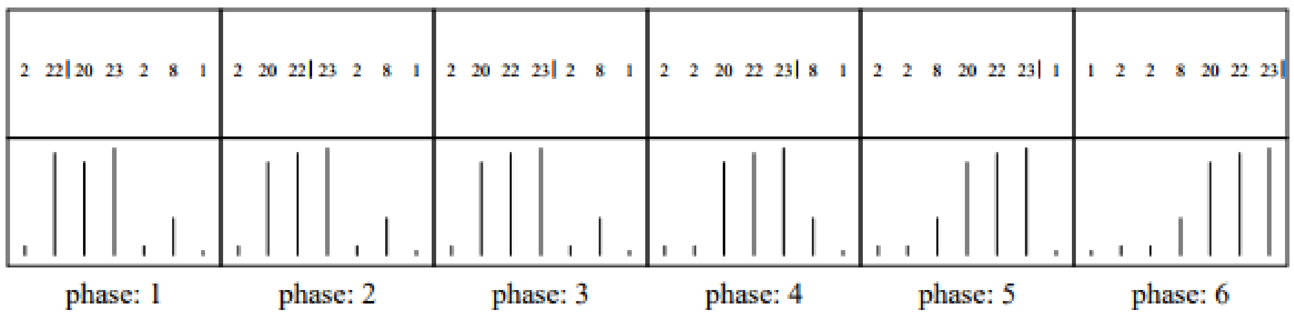
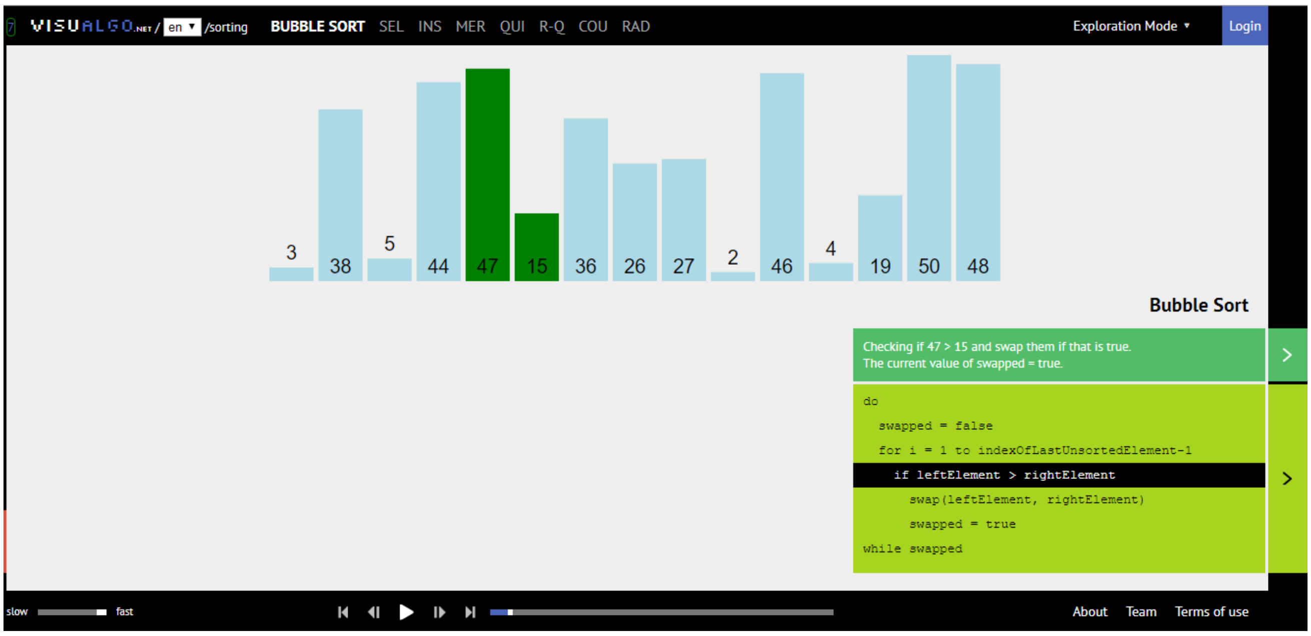


Figure 2.1: A sequence of stills from an insertion sort algorithm

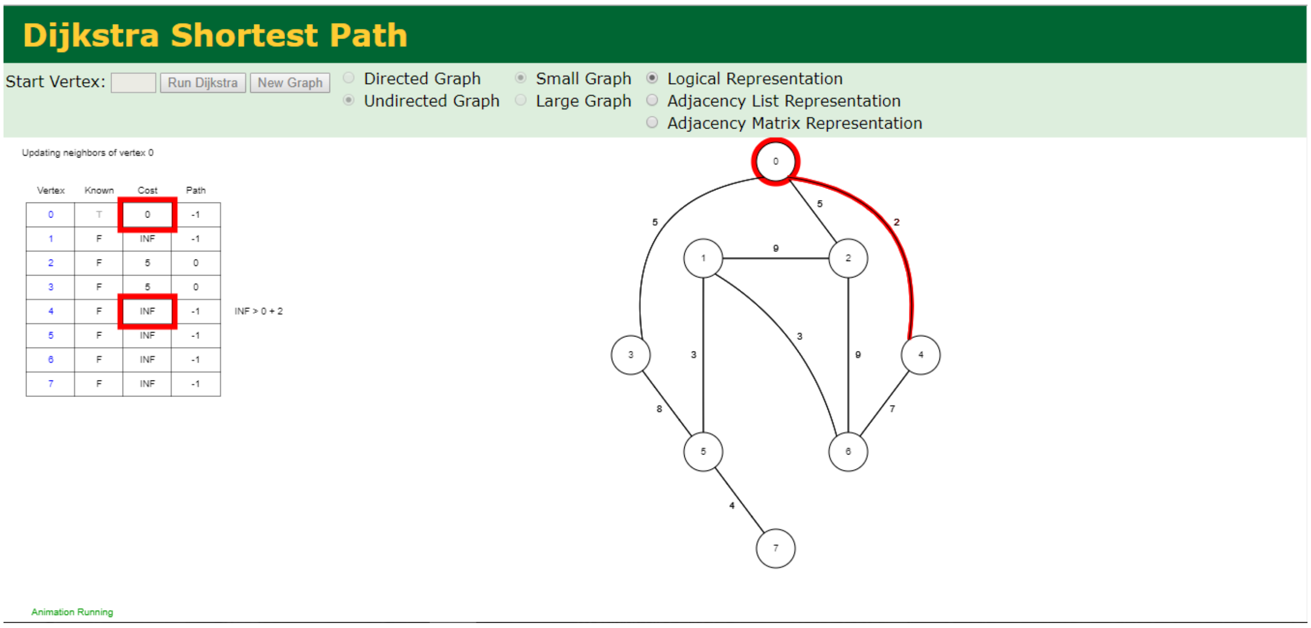


Bubble Sort

Checking if 47 > 15 and swap them if that is true.
The current value of swapped = true.

```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true
while swapped
```

Figure 2.2a: Examples of existing algorithm animation software - VisuAlgo



Directed Graph [✓ Undirected Graph]
[✓ Small Graph] Large Graph
[✓ Logical Representation] Adjacency List Representation Adjacency Matrix Representation

Updating neighbors of vertex 0

Vertex	Known	Cost	Path
0	T	0	-1
1	F	INF	-1
2	F	5	0
3	F	5	0
4	F	INF	-1
5	F	INF	-1
6	F	INF	-1
7	F	INF	-1

INF > 0 + 2

Animation Running

Figure 2.2b: Examples of existing algorithm animation software - USF's Dijkstra's algorithm animation

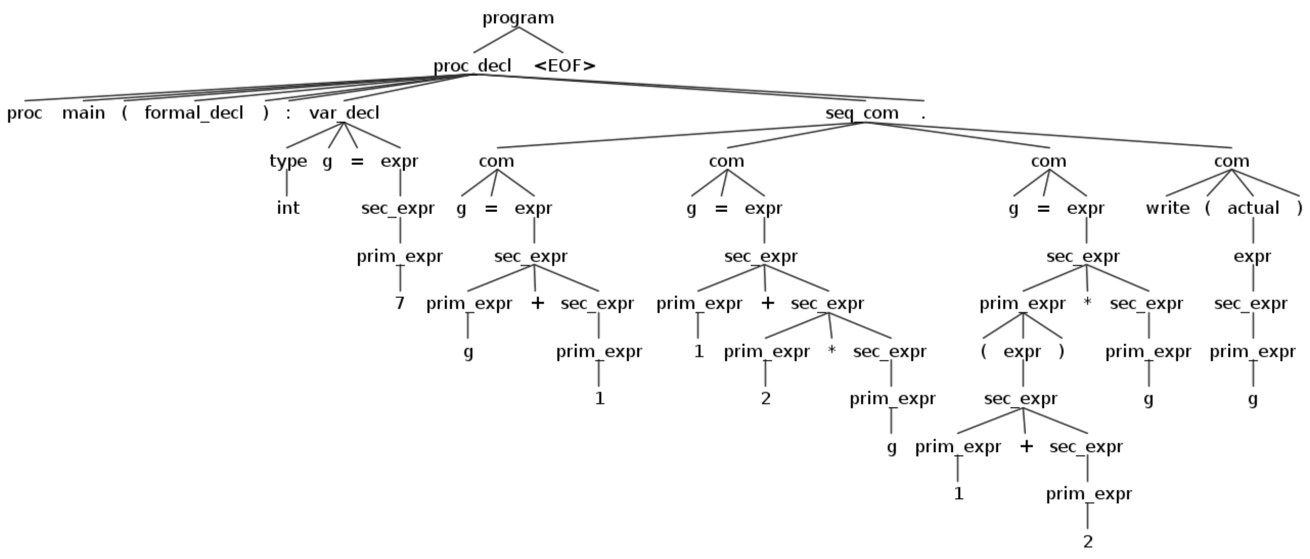


Figure 3.1a: Parse tree and AST of the same hypothetical Fun program - Parse tree

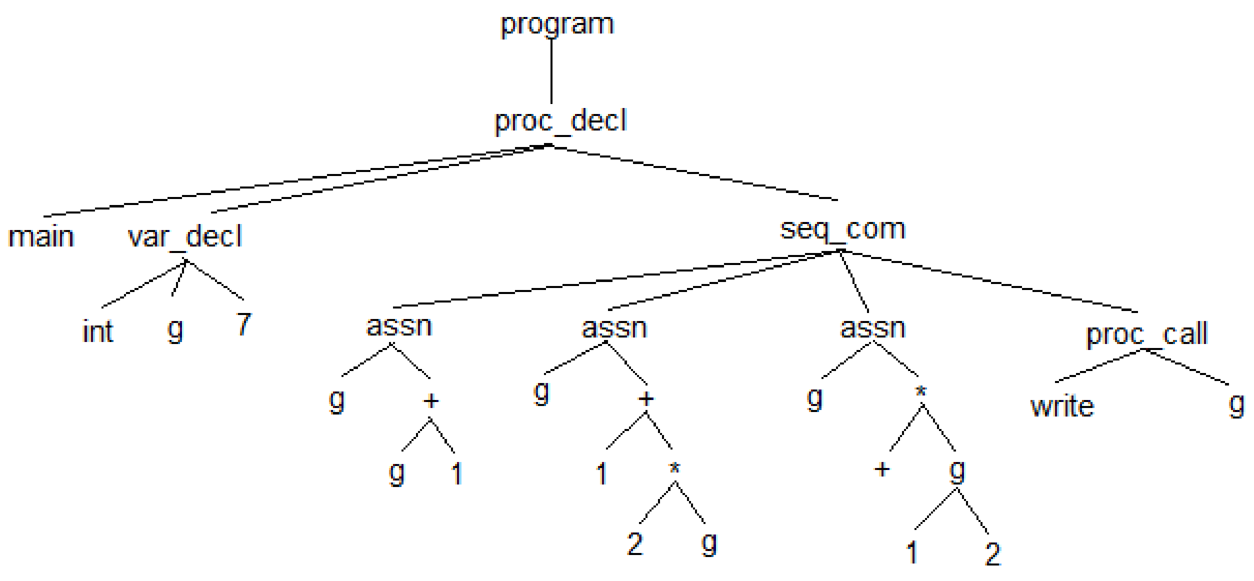


Figure 3.1b: Parse tree and AST of the same hypothetical Fun program - AST

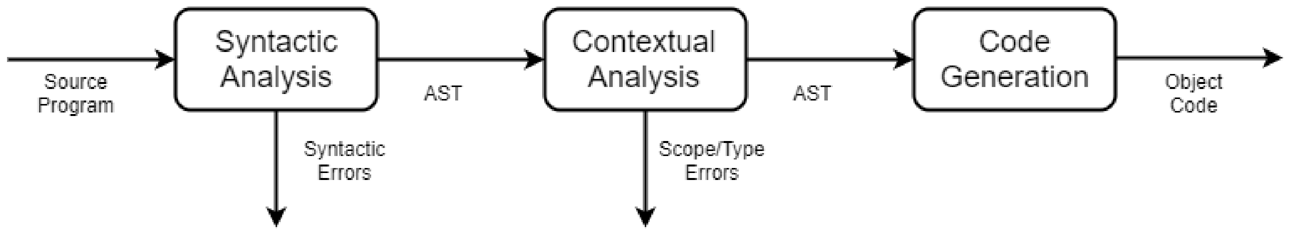
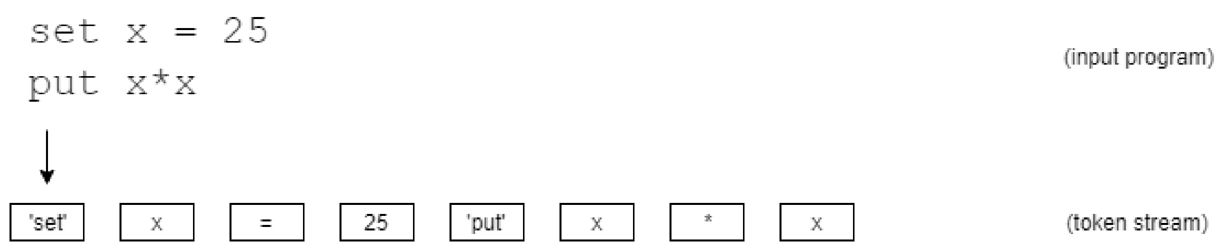


Figure 3.2: Typical compilation pipeline



```
set x = 25
put x*x
```

Figure 3.3: Lexing of source code into a stoken stream

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

Figure 3.4: Type table

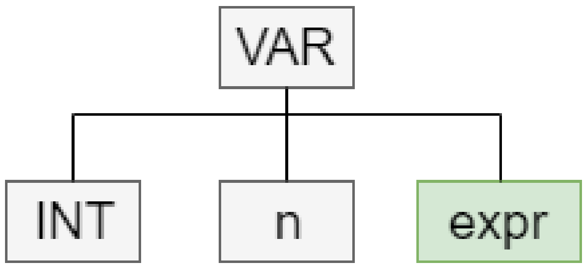


Figure 3.5a: ASTs demonstrating the declaration and assignment of a variable `n` (assuming global scope) - Variable declaration

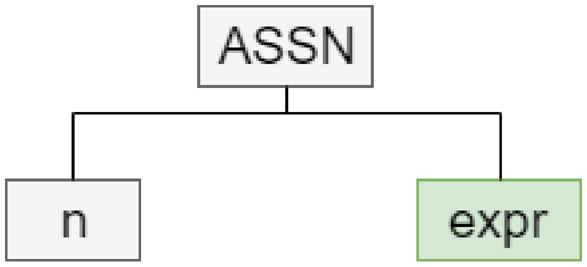
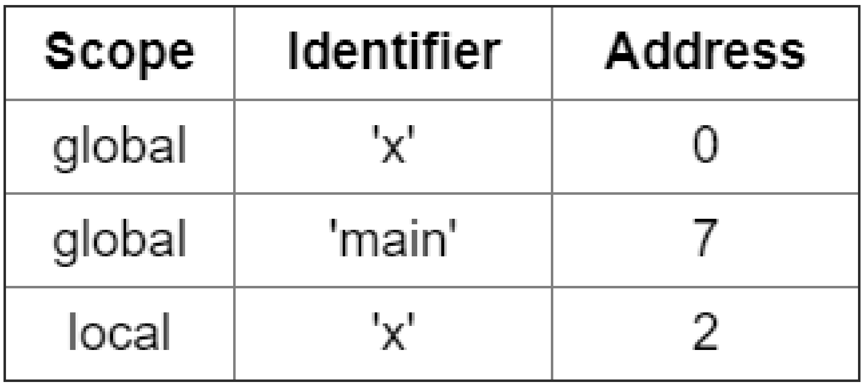


Figure 3.5b: ASTs demonstrating the declaration and assignment of a variable `n` (assuming global scope) - Variable assignment



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

Figure 3.6: Address table

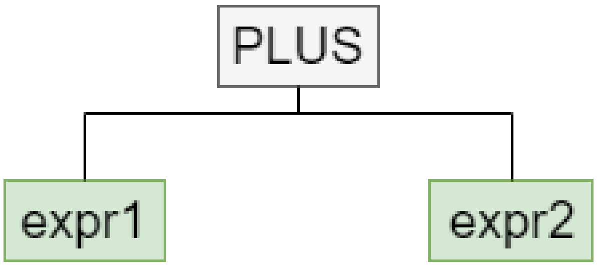
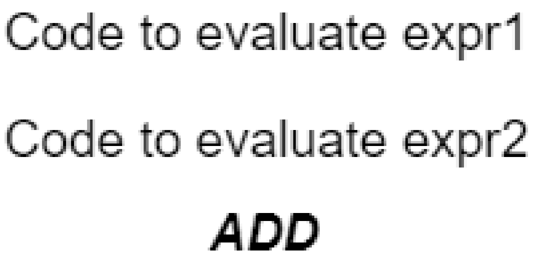


Figure 3.7a: AST and code template of a `PLUS` construct - AST

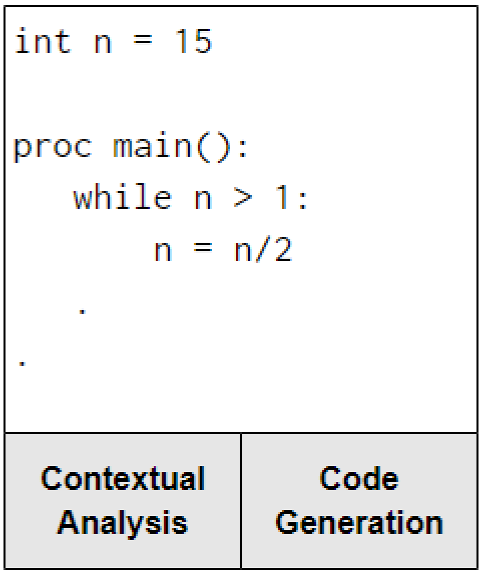


Code to evaluate expr1

Code to evaluate expr2

ADD

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



```
int n = 15

proc main():
   while n > 1:
       n = n/2
   .
.
```

Figure 5.1: Code editor wireframe

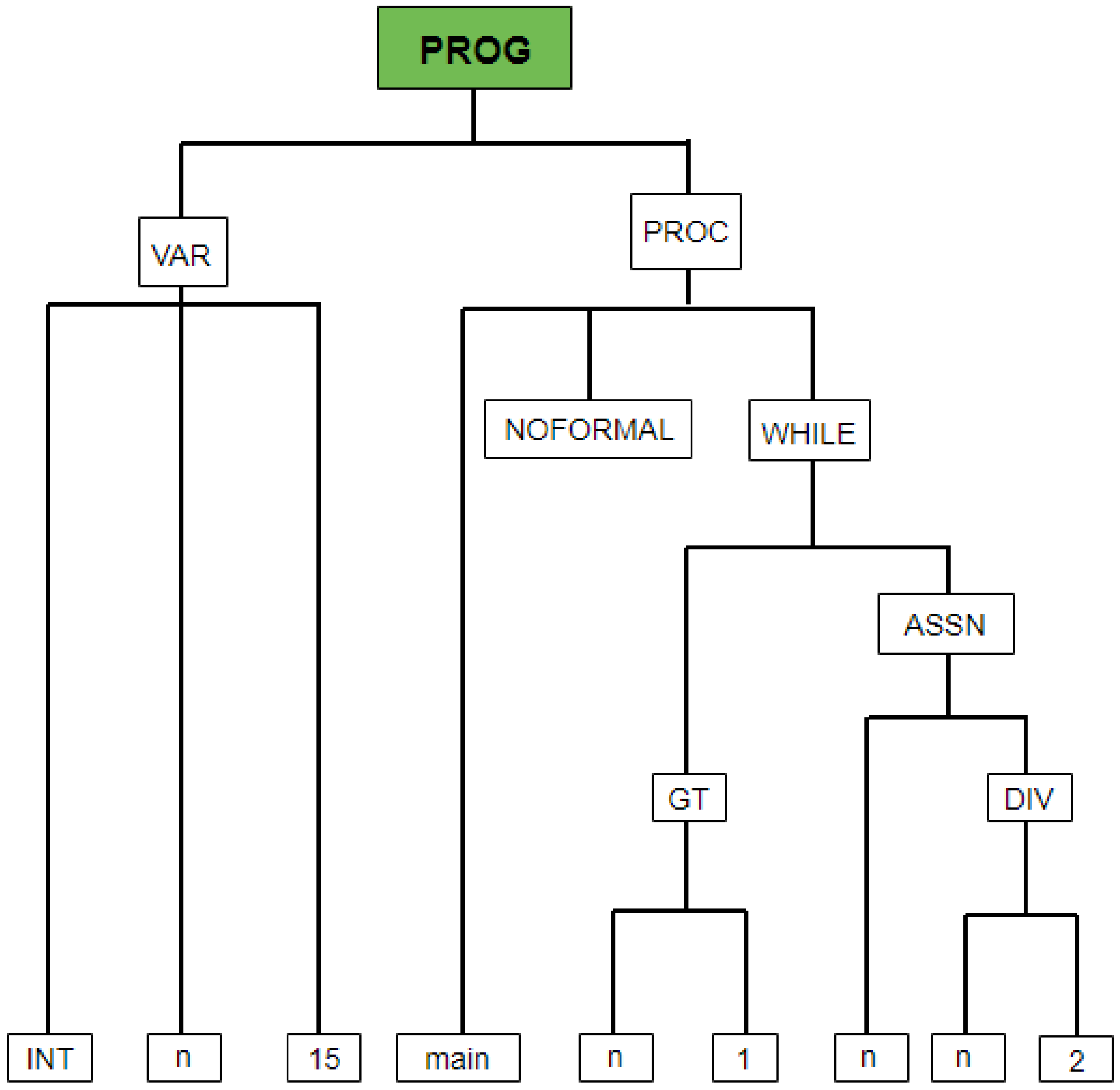


Figure 5.2a: AST wireframes - Step 1 of animation

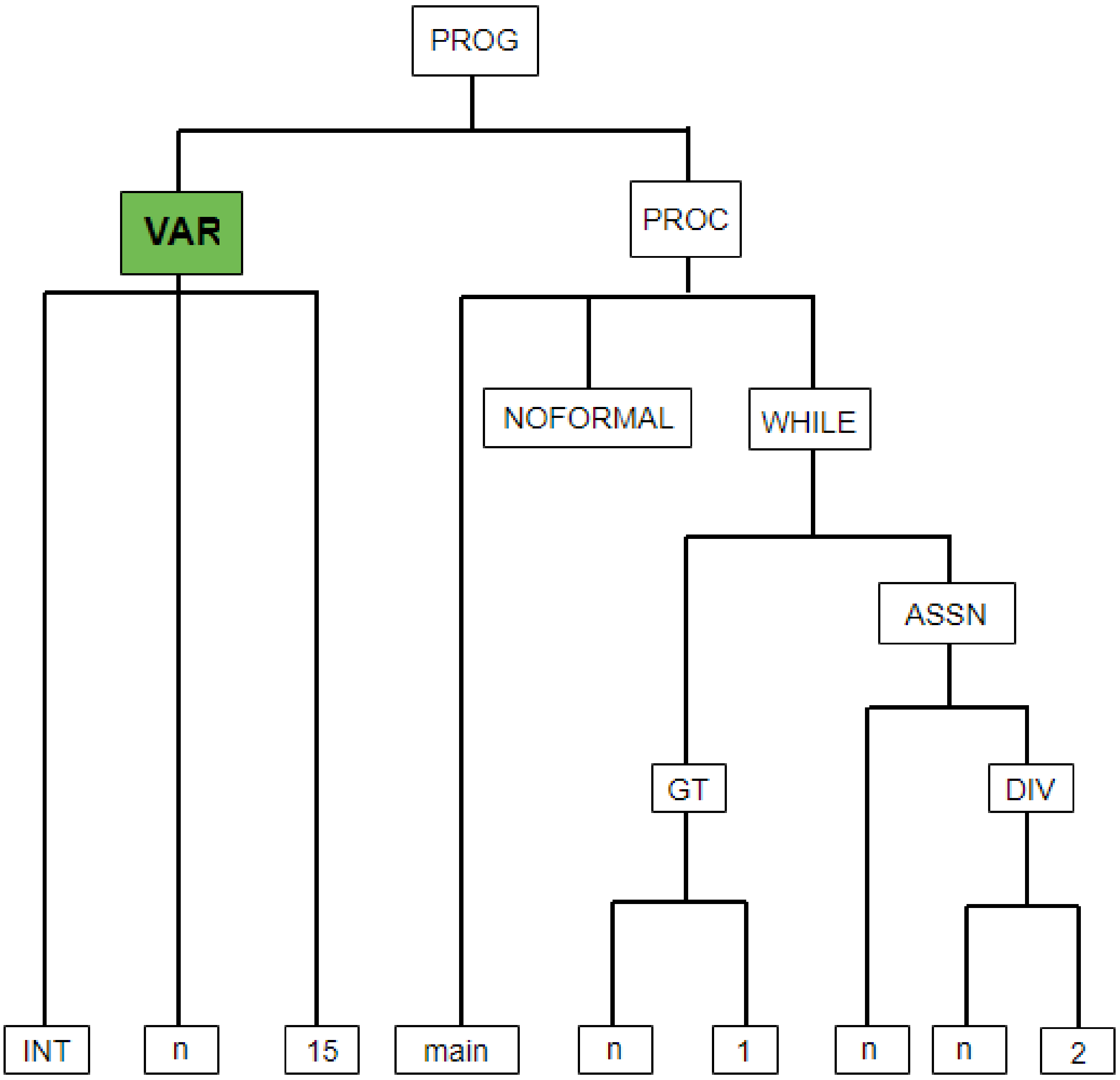


Figure 5.2b: AST wireframes - Step 2 of animation

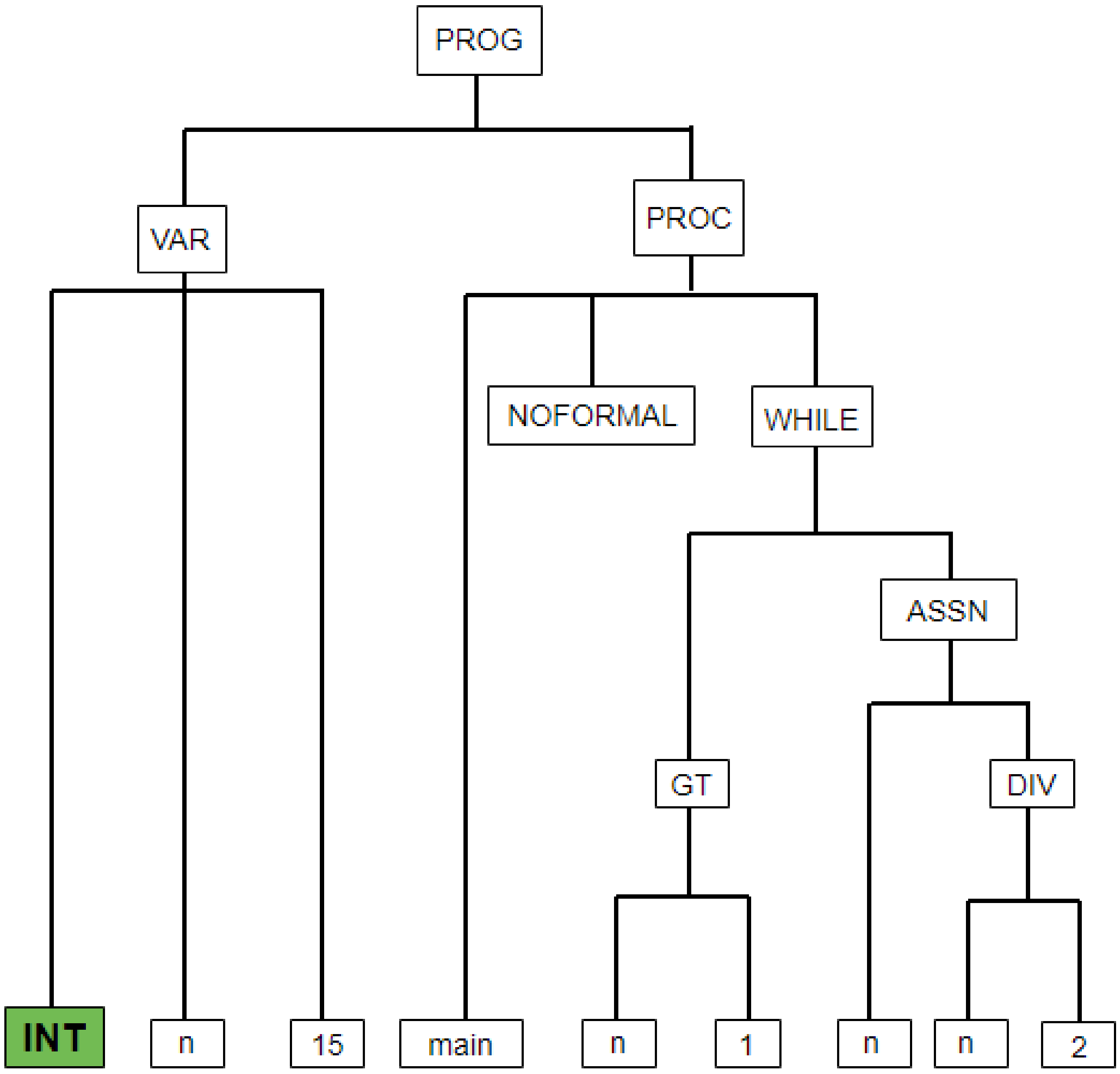
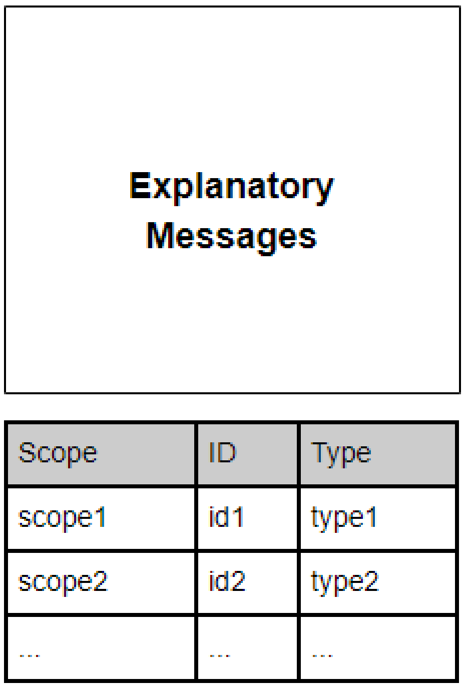
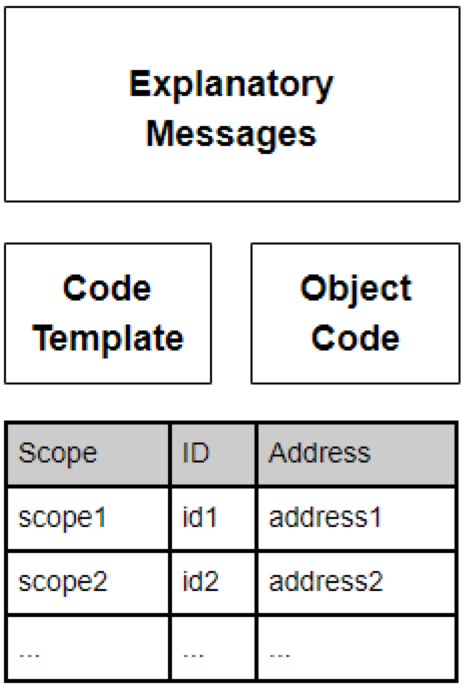


Figure 5.2c: AST wireframes - Step 3 of animation



Scope	ID	Type
scope1	id1	type1
scope2	id2	type2
...	...	...

Figure 5.3a: Augmentations wireframe - Contextual analysis augmentations

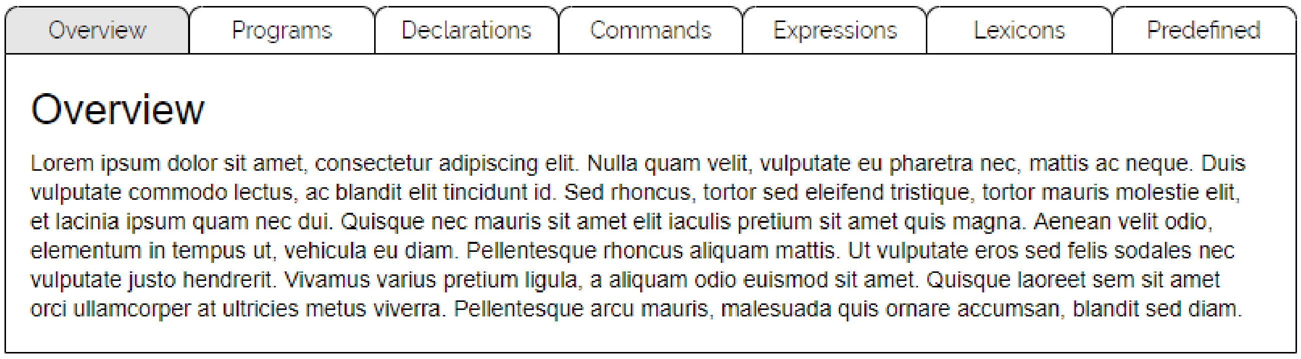


Scope	ID	Address
scope1	id1	address1
scope2	id2	address2
...	...	...

Figure 5.3b: Augmentations wireframe - Code generation augmentations

Figure 5.4a: Playback controls wireframe

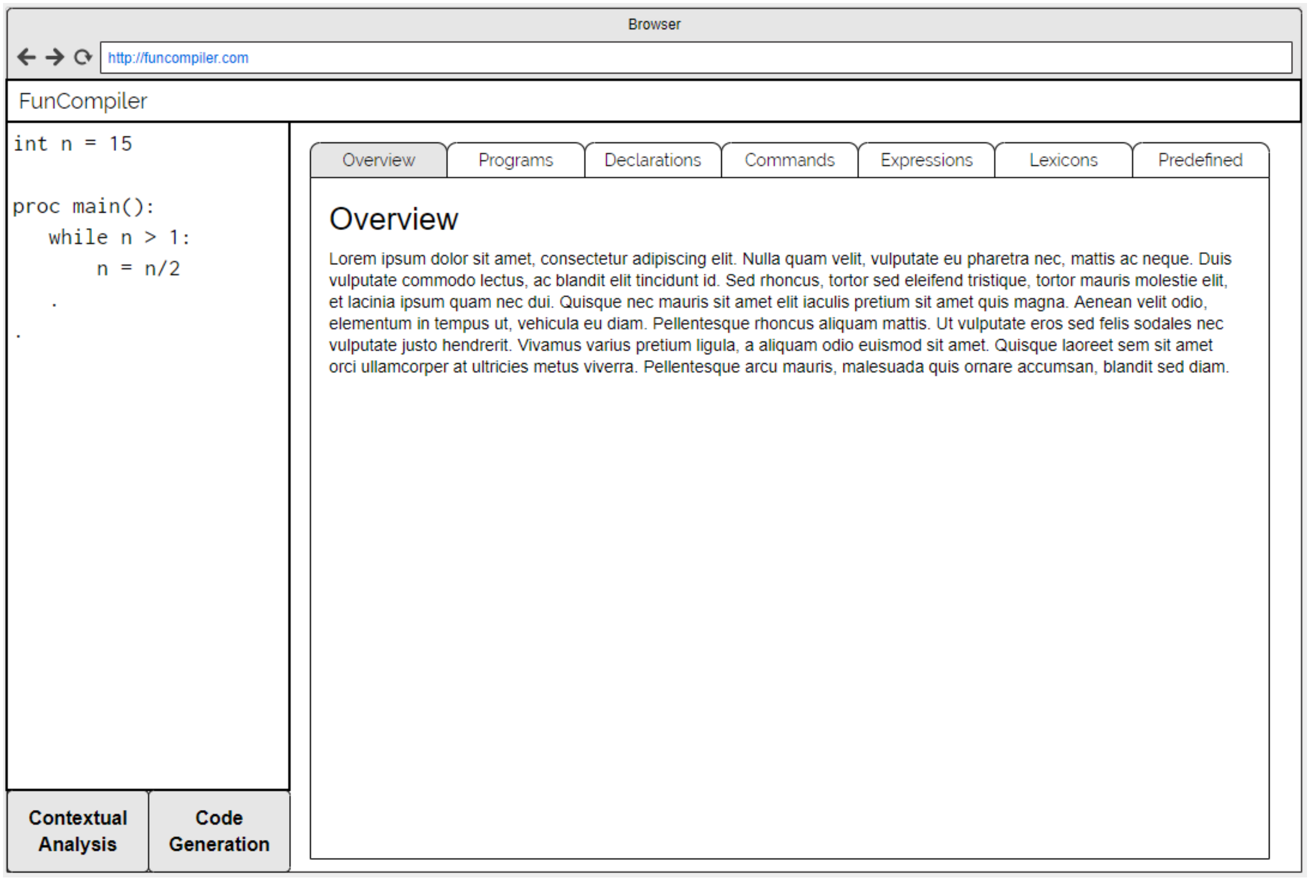
Figure 5.4b: Playback controls wireframe



Overview

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla quam velit, vulputate eu pharetra nec, mattis ac neque. Duis vulputate commodo lectus, ac blandit elit tincidunt id. Sed rhoncus, tortor sed eleifend tristique, tortor mauris molestie elit, et lacinia ipsum quam nec dui. Quisque nec mauris sit amet elit iaculis pretium sit amet quis magna. Aenean velit odio, elementum in tempus ut, vehicula eu diam. Pellentesque rhoncus aliquam mattis. Ut vulputate eros sed felis sodales nec vulputate justo hendrerit. Vivamus varius pretium ligula, a aliquam odio euismod sit amet. Quisque laoreet sem sit amet orci ullamcorper at ultricies metus viverra. Pellentesque arcu mauris, malesuada quis ornare accumsan, blandit sed diam.

Figure 5.5: Fun specification wireframe

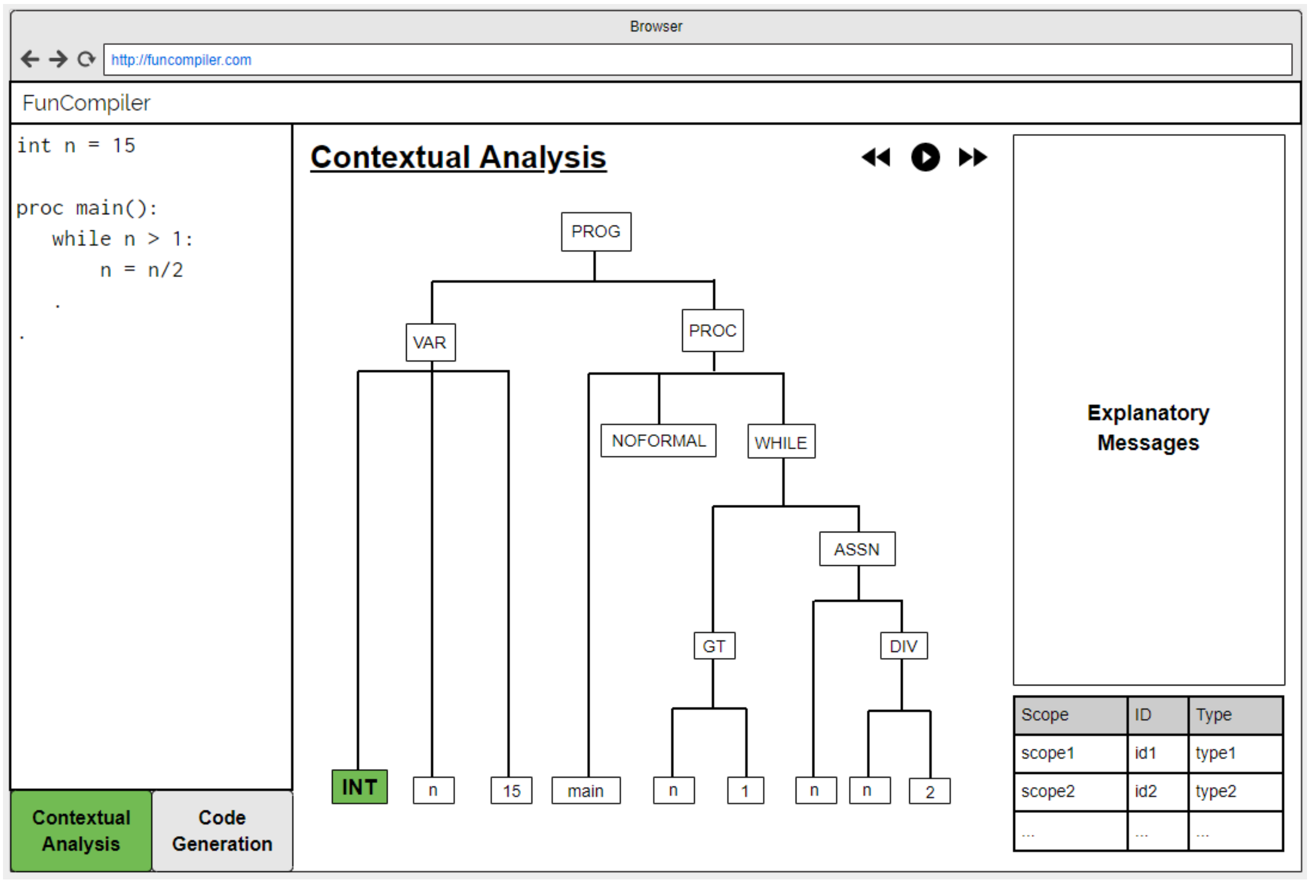


```
int n = 15

proc main():
   while n > 1:
       n = n/2
   .
.
```

Overview

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla quam velit, vulputate eu pharetra nec, mattis ac neque. Duis vulputate commodo lectus, ac blandit elit tincidunt id. Sed rhoncus, tortor sed eleifend tristique, tortor mauris molestie elit, et lacinia ipsum quam nec dui. Quisque nec mauris sit amet elit iaculis pretium sit amet quis magna. Aenean velit odio, elementum in tempus ut, vehicula eu diam. Pellentesque rhoncus aliquam mattis. Ut vulputate eros sed felis sodales nec vulputate justo hendrerit. Vivamus varius pretium ligula, a aliquam odio euismod sit amet. Quisque laoreet sem sit amet orci ullamcorper at ultricies metus viverra. Pellentesque arcu mauris, malesuada quis ornare accumsan, blandit sed diam.

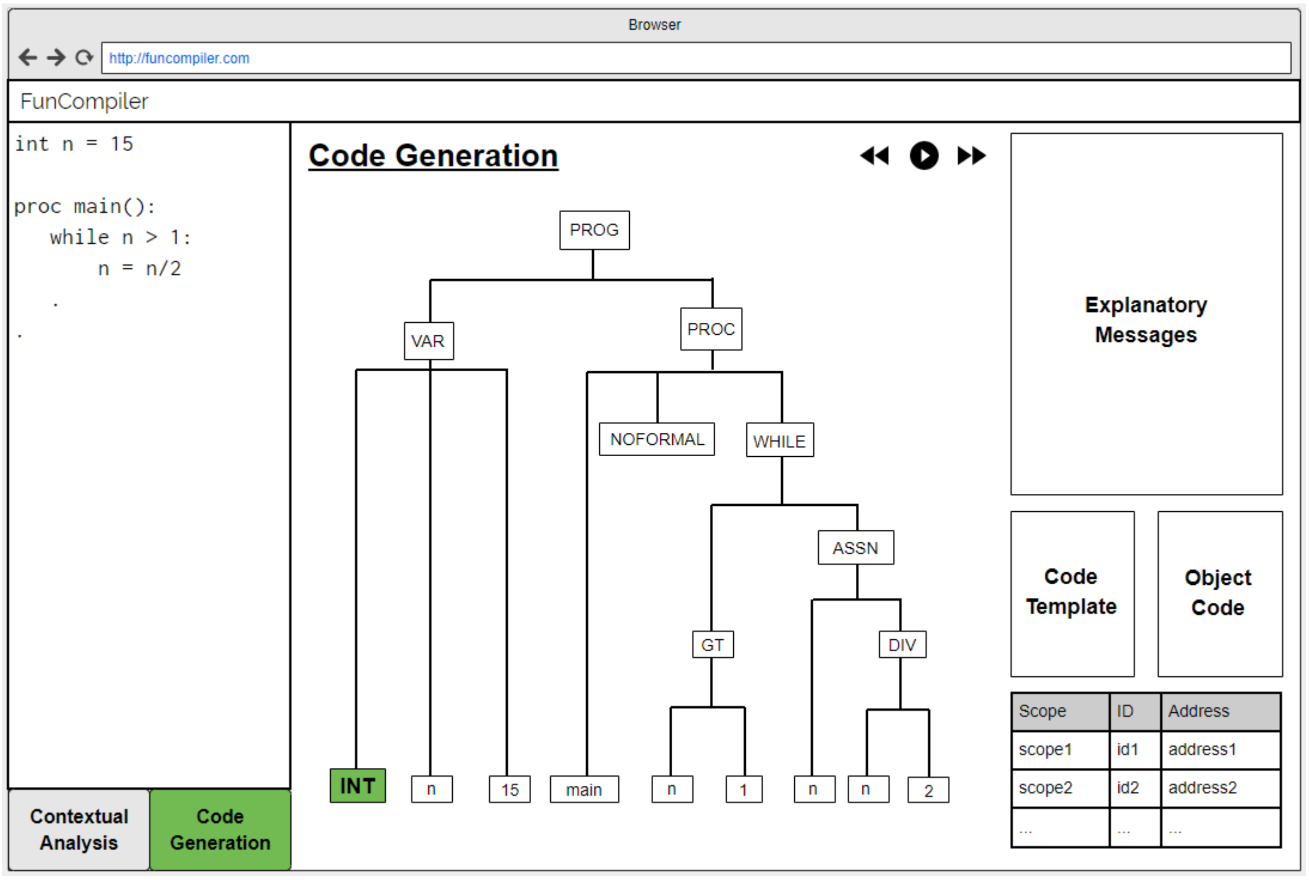
Figure 5.6a: The three main sections of the FunCompiler - Landing page wireframe



```
int n = 15

proc main():
   while n > 1:
       n = n/2
   .
.
```

Scope	ID	Type
scope1	id1	type1
scope2	id2	type2
...	...	...

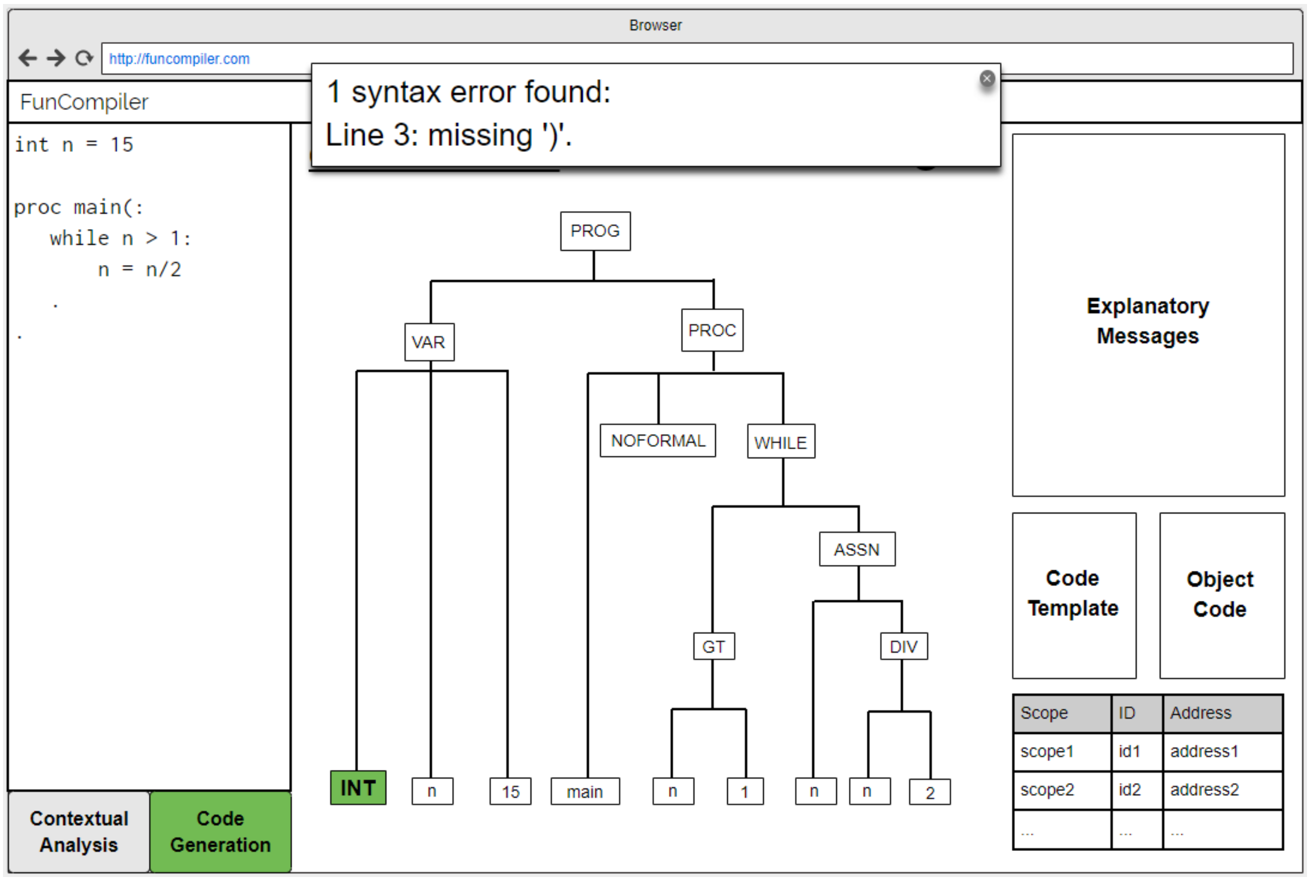
Figure 5.6b: The three main sections of the FunCompiler - Contextual analysis animation wireframe



```
int n = 15

proc main():
   while n > 1:
       n = n/2
   .
.
```

Scope	ID	Address
scope1	id1	address1
scope2	id2	address2
...	...	...

Figure 5.6c: The three main sections of the FunCompiler - Code generation animation wireframe



```
int n = 15

proc main(:
   while n > 1:
      n = n/2
      .
.
```

1 syntax error found:
Line 3: missing ')'.

Scope	ID	Address
scope1	id1	address1
scope2	id2	address2
...	...	...

Figure 5.7: Syntax error wireframe

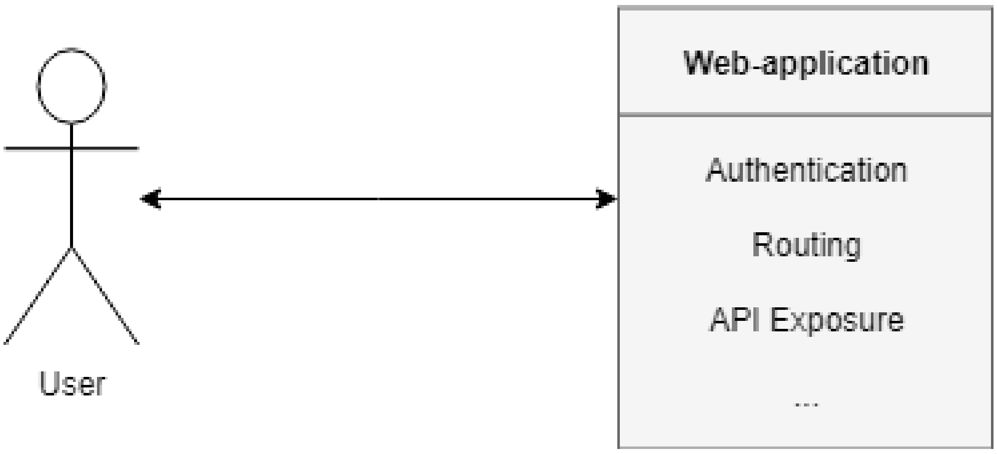


Figure 6.1a: Monolithic vs. micro-service architecture - Monolithic architecture

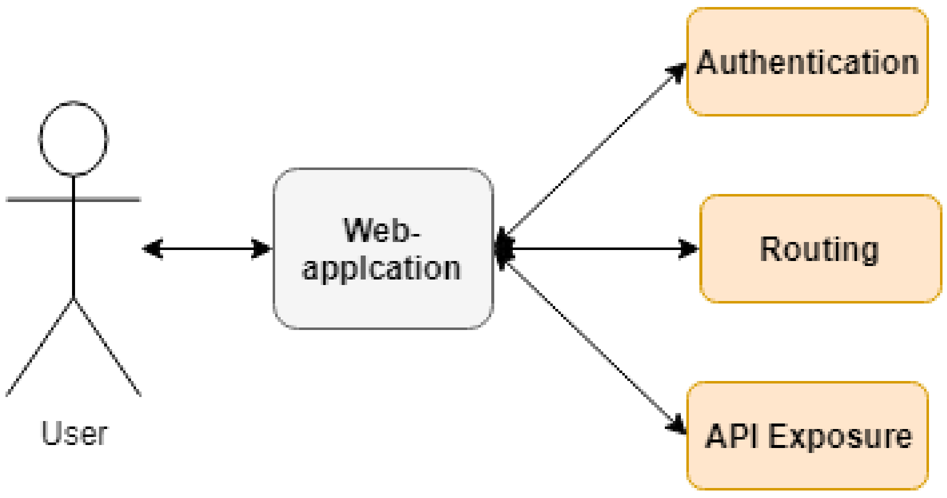


Figure 6.1b: Monolithic vs. micro-service architecture - Micro-service architecture

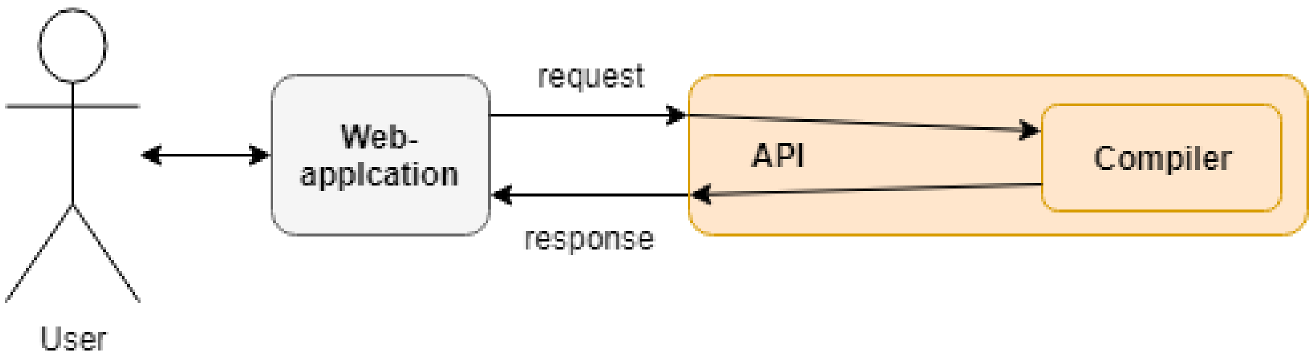


Figure 6.2: Web-application communicating with the compiler via an API

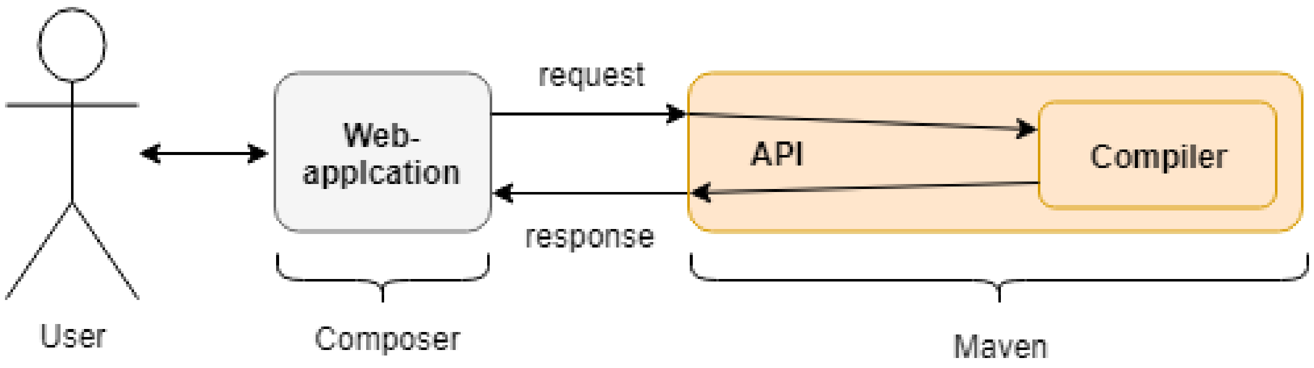


Figure 6.3: Current system infrastructure

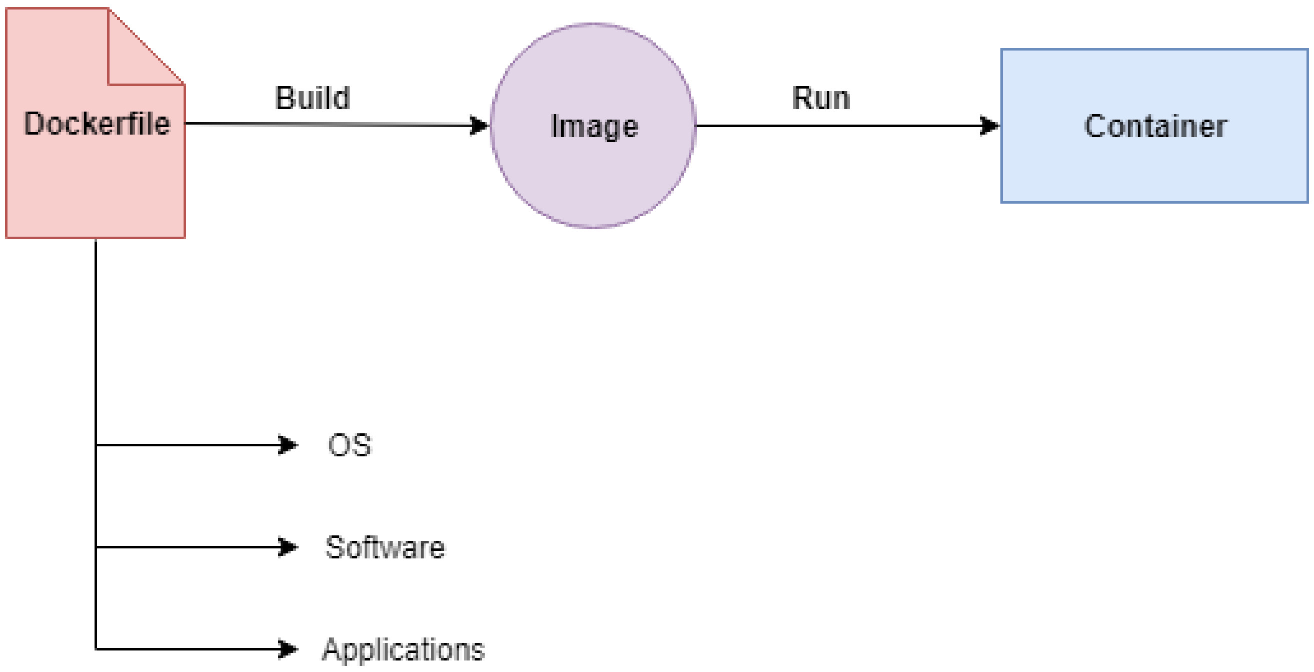


Figure 6.4: Creating a Docker container from an image

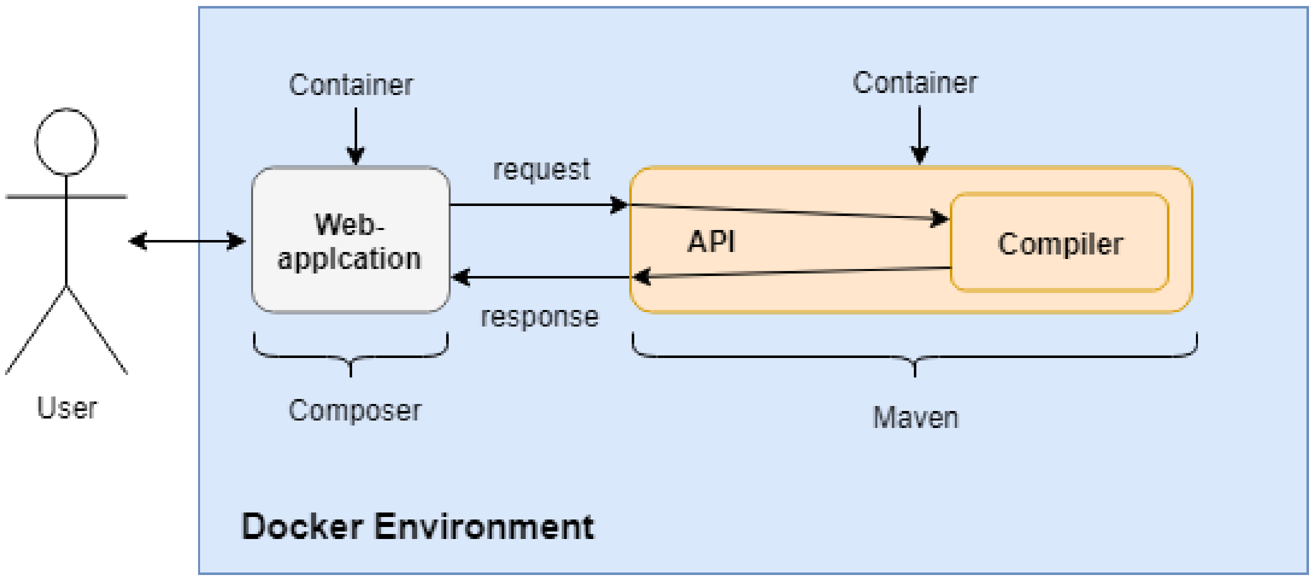


Figure 6.5: Final infrastructure of the Funcompiler

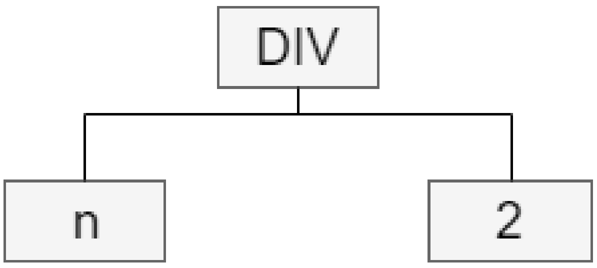


Figure 7.1: The visual representation of an AST as derived from Listing 7.3

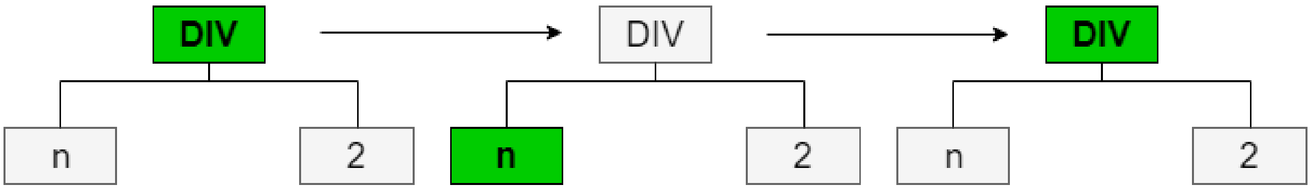


Figure 7.2: The animation of the AST as dictated by the Listings 7.3 and 7.4

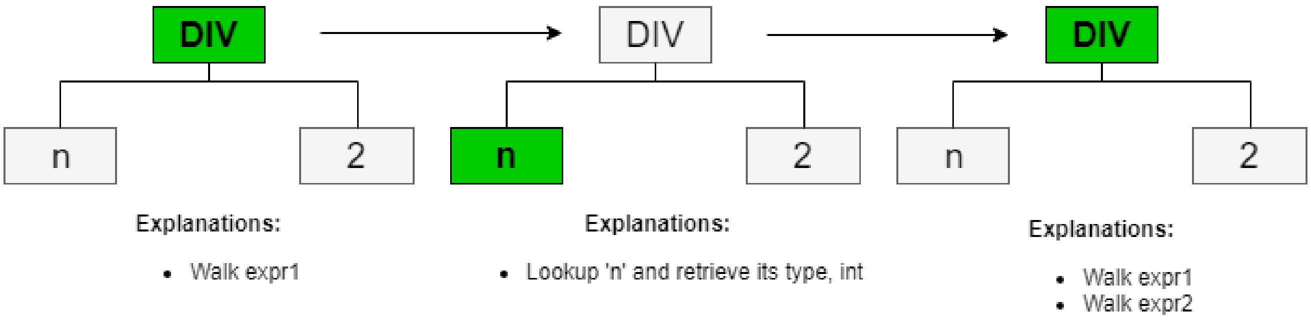
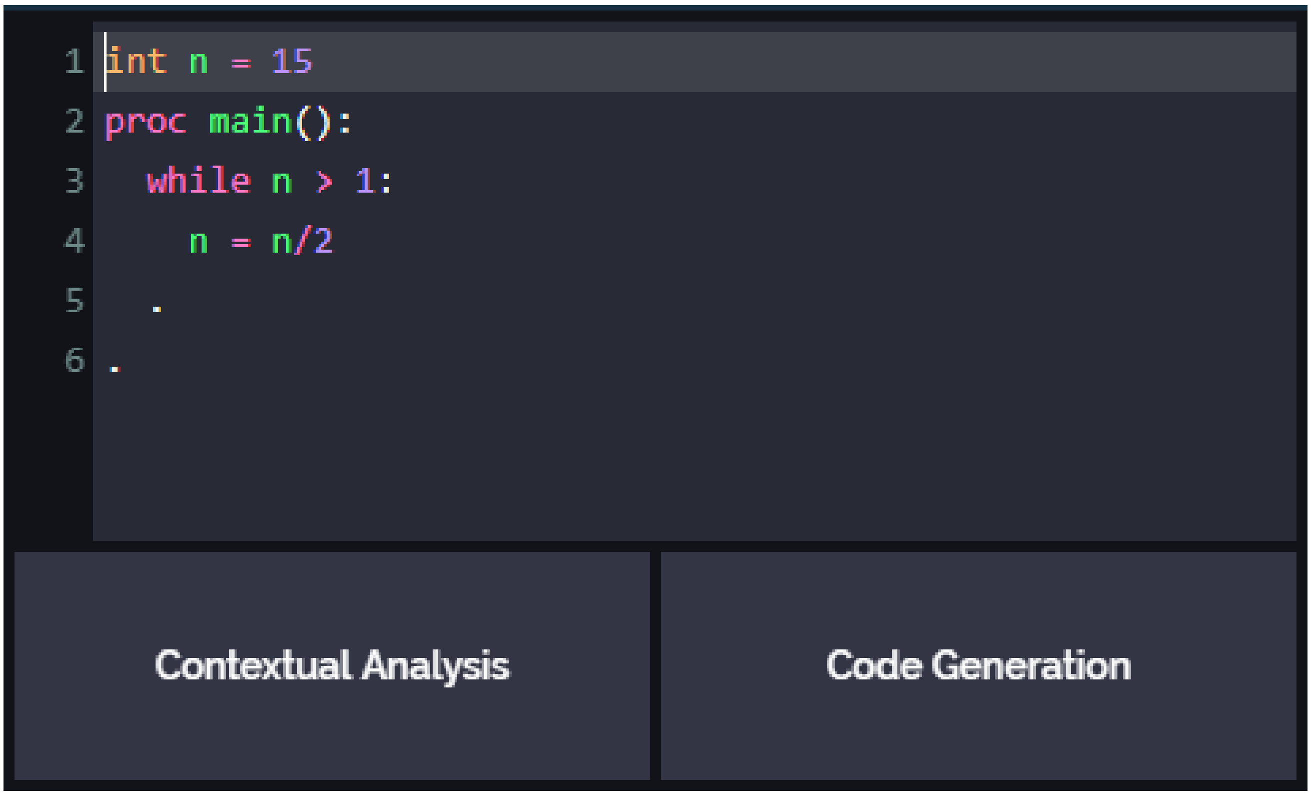


Figure 7.3: The animation of the AST as dictated by the Listings 7.3 and 7.4 with explanatory messages added



```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

Figure 7.4: Implementation of the code editor

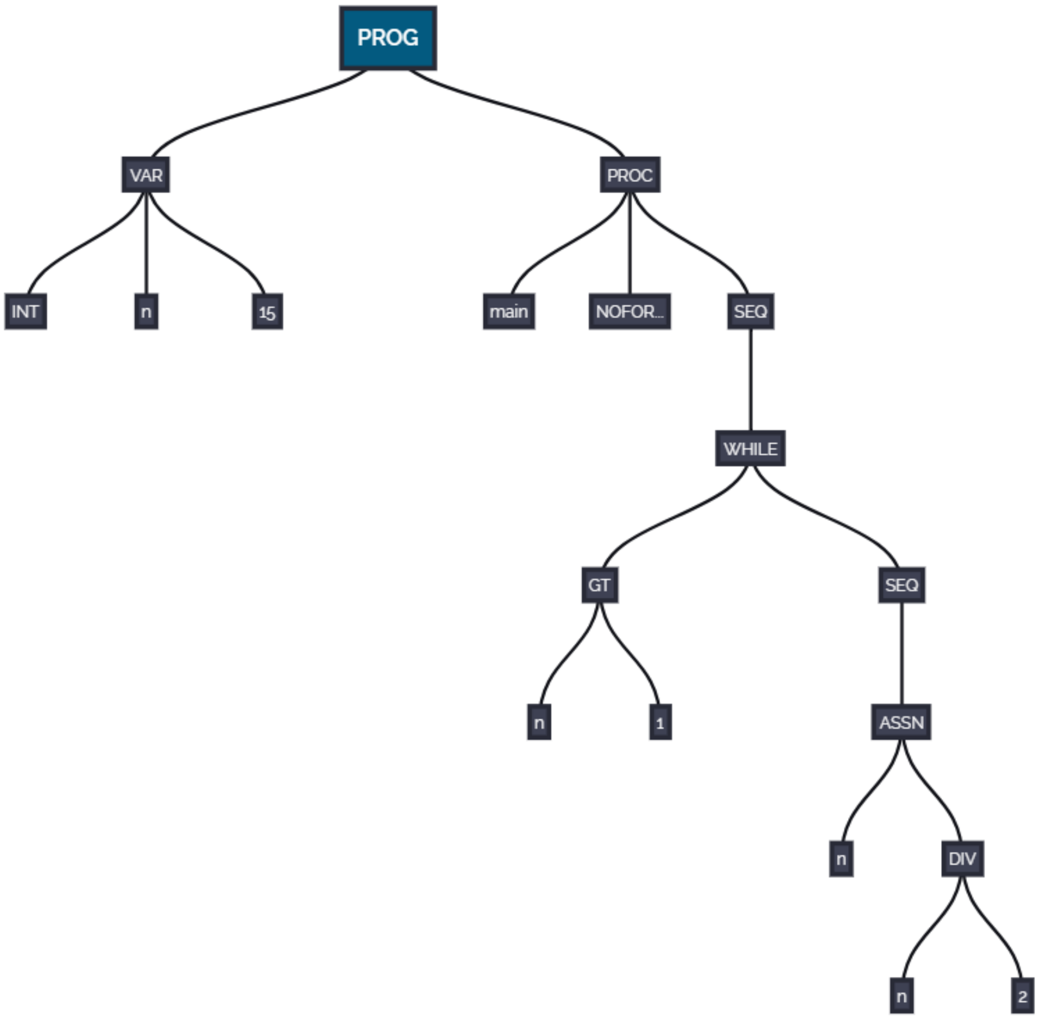


Figure 7.5a: SVG representations of different Fun programs generated using D3

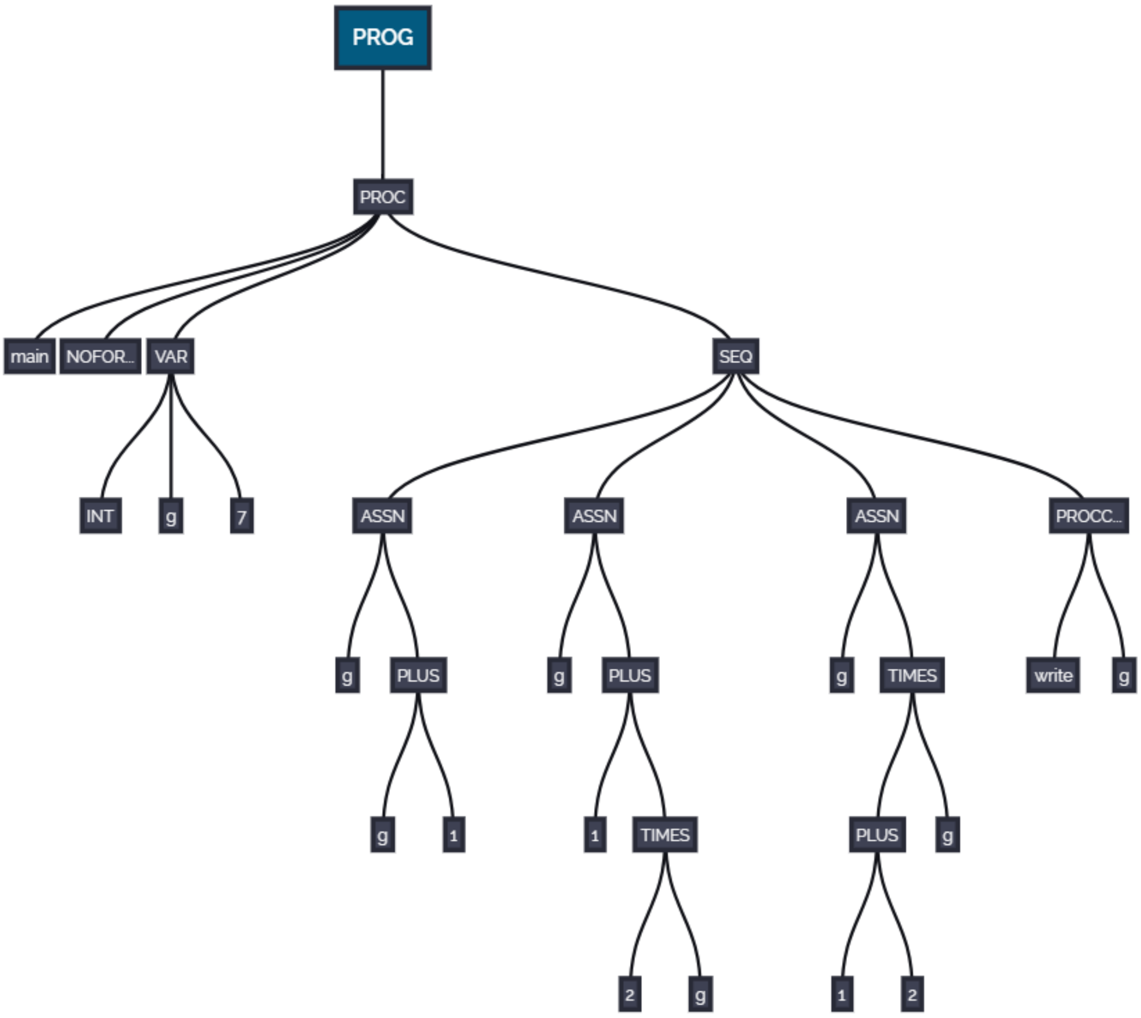


Figure 7.5b: SVG representations of different Fun programs generated using D3

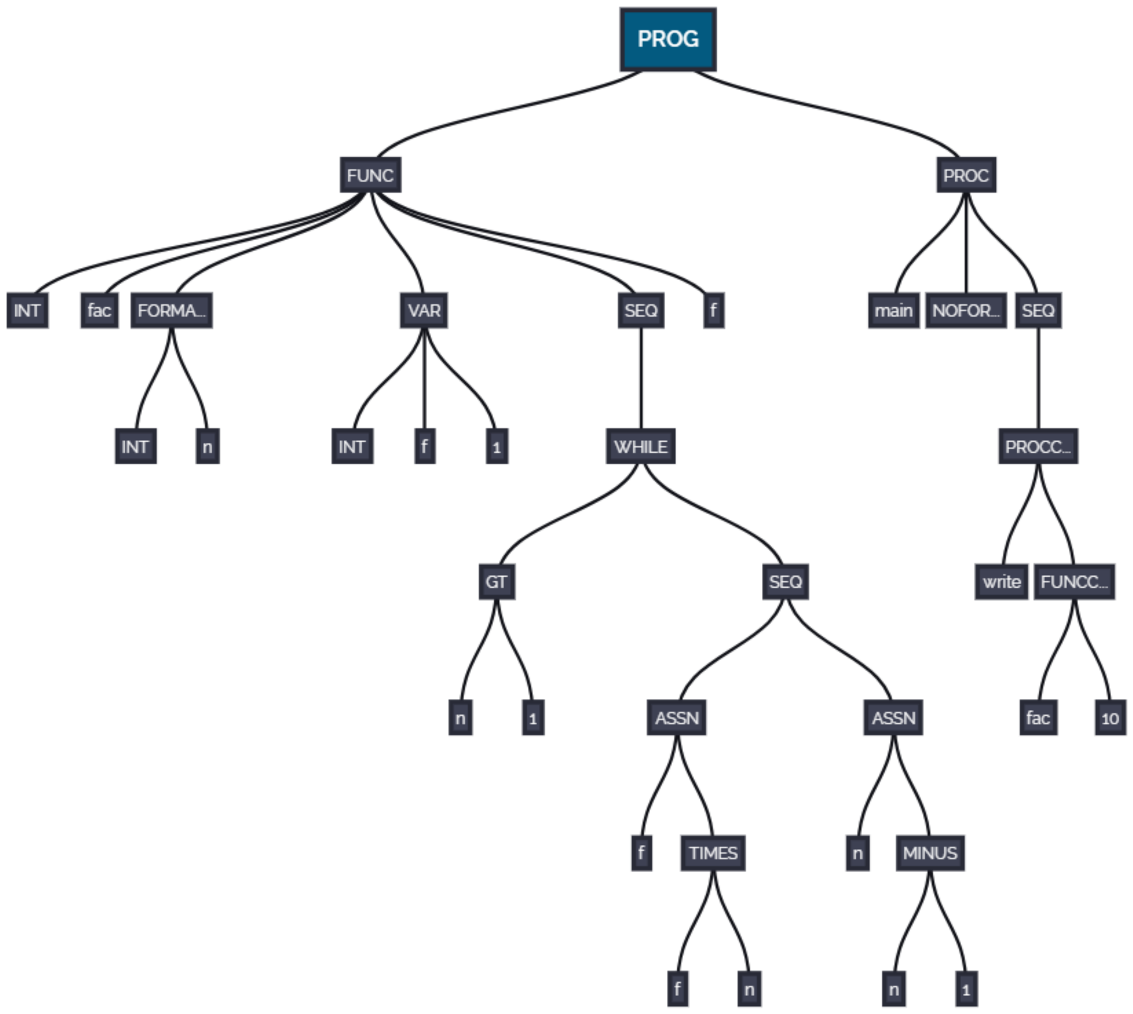


Figure 7.5c: SVG representations of different Fun programs generated using D3

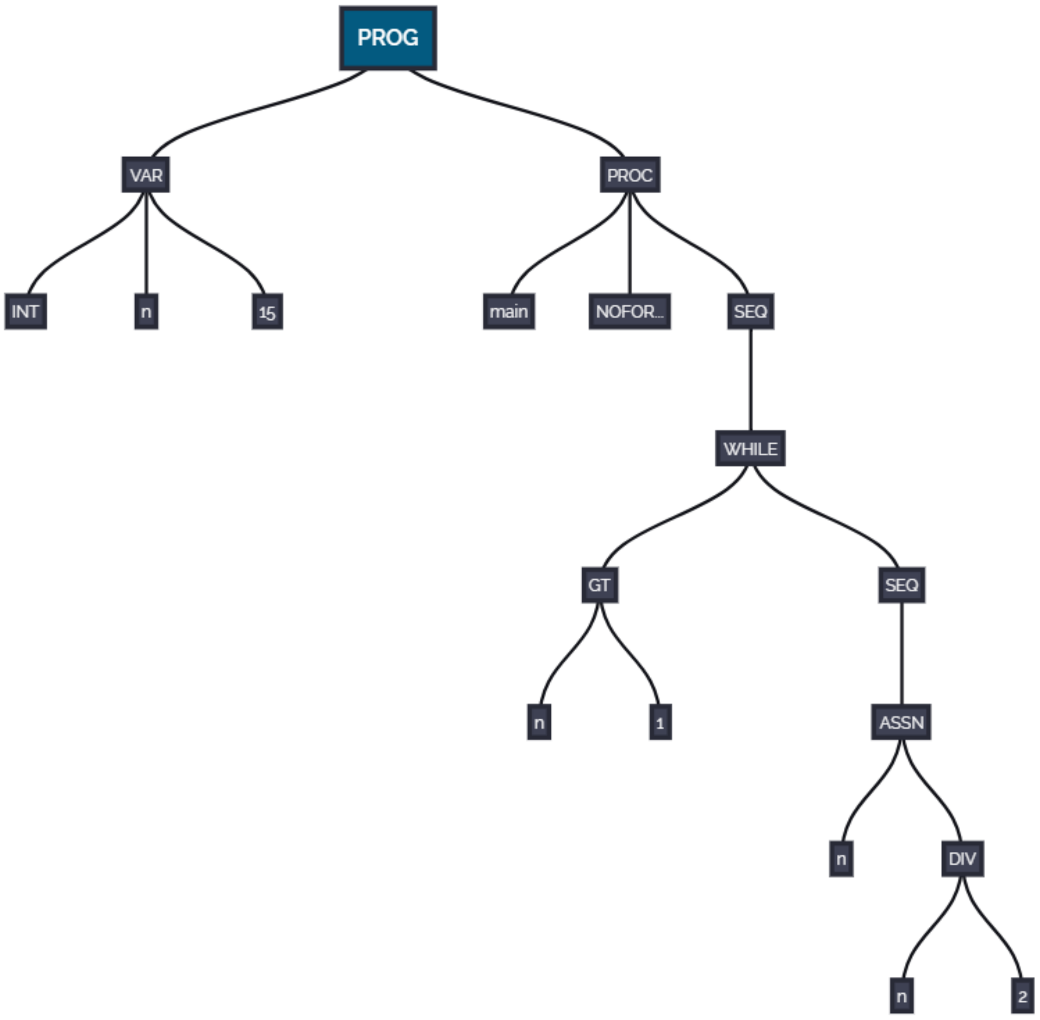


Figure 7.6a: The first three steps of a Fun program's animation - Step 1 of animation

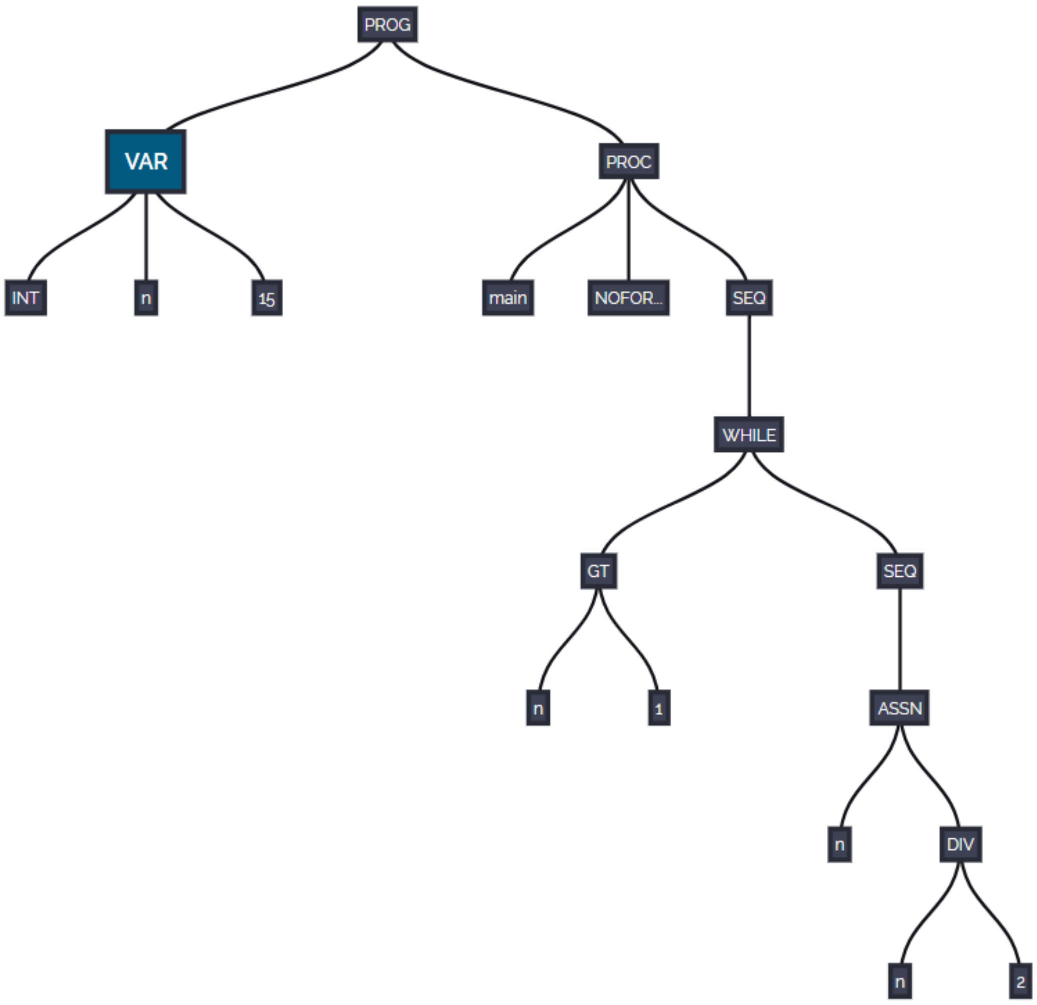


Figure 7.6b: The first three steps of a Fun program's animation - Step 2 of animation

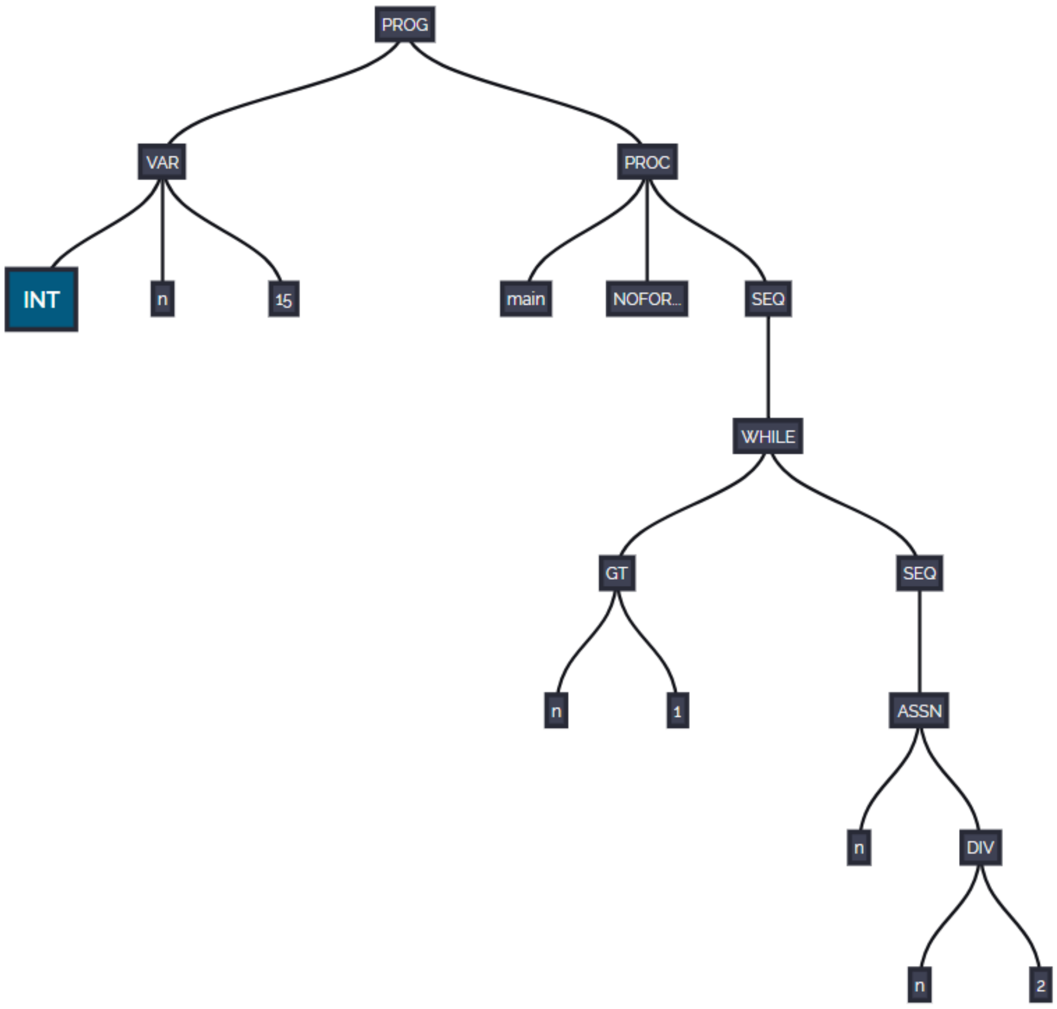
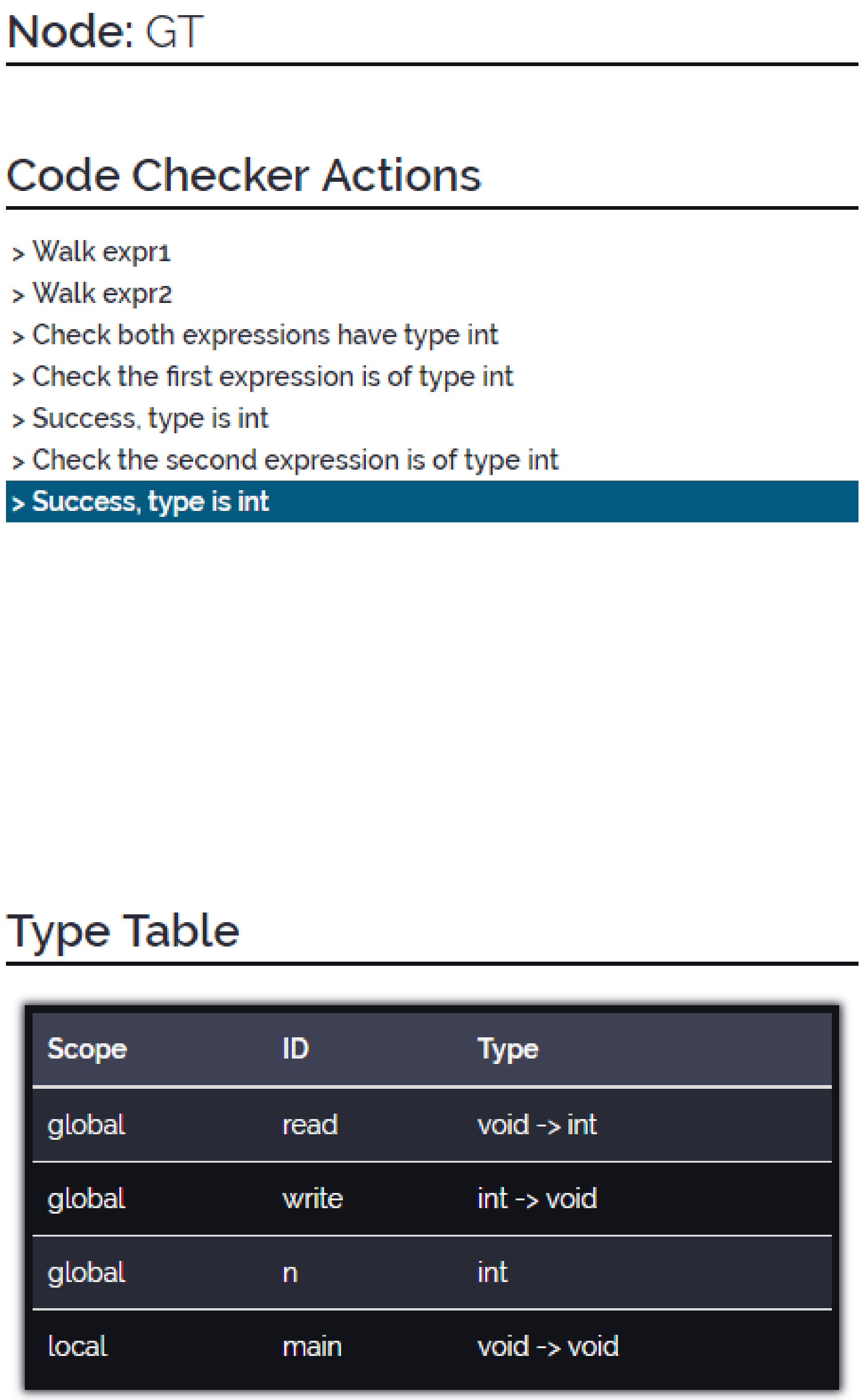


Figure 7.6c: The first three steps of a Fun program's animation - Step 3 of animation



Node: GT

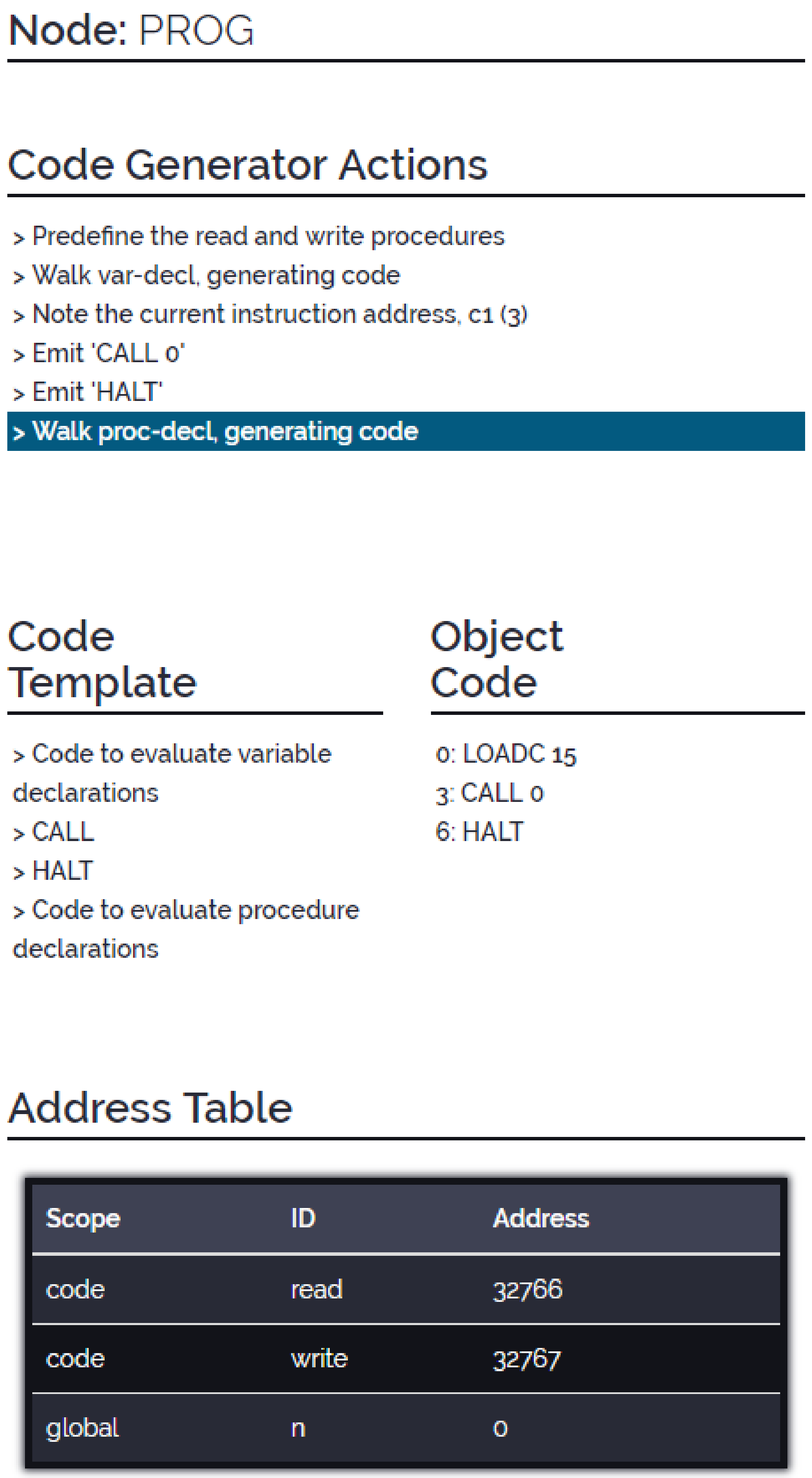
Code Checker Actions

> 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

Figure 7.7a: Implementation of the augmentations - 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

Figure 7.7b: Implementation of the augmentations - Code generation augmentations

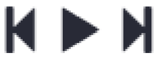


Figure 7.8a: Implementation of the playback controls

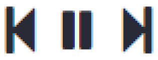
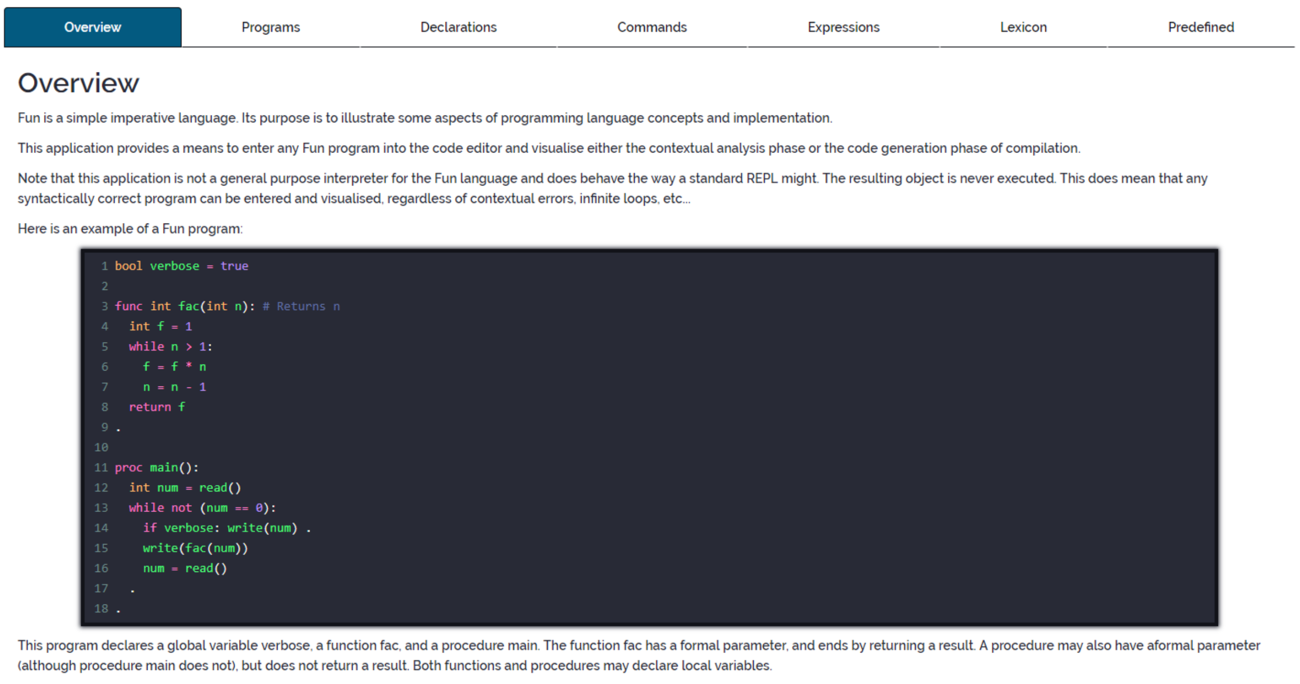


Figure 7.8b: Implementation of the playback controls



Overview

Fun is a simple imperative language. Its purpose is to illustrate some aspects of programming language concepts and implementation.

This application provides a means to enter any Fun program into the code editor and visualise either the contextual analysis phase or the code generation phase of compilation.

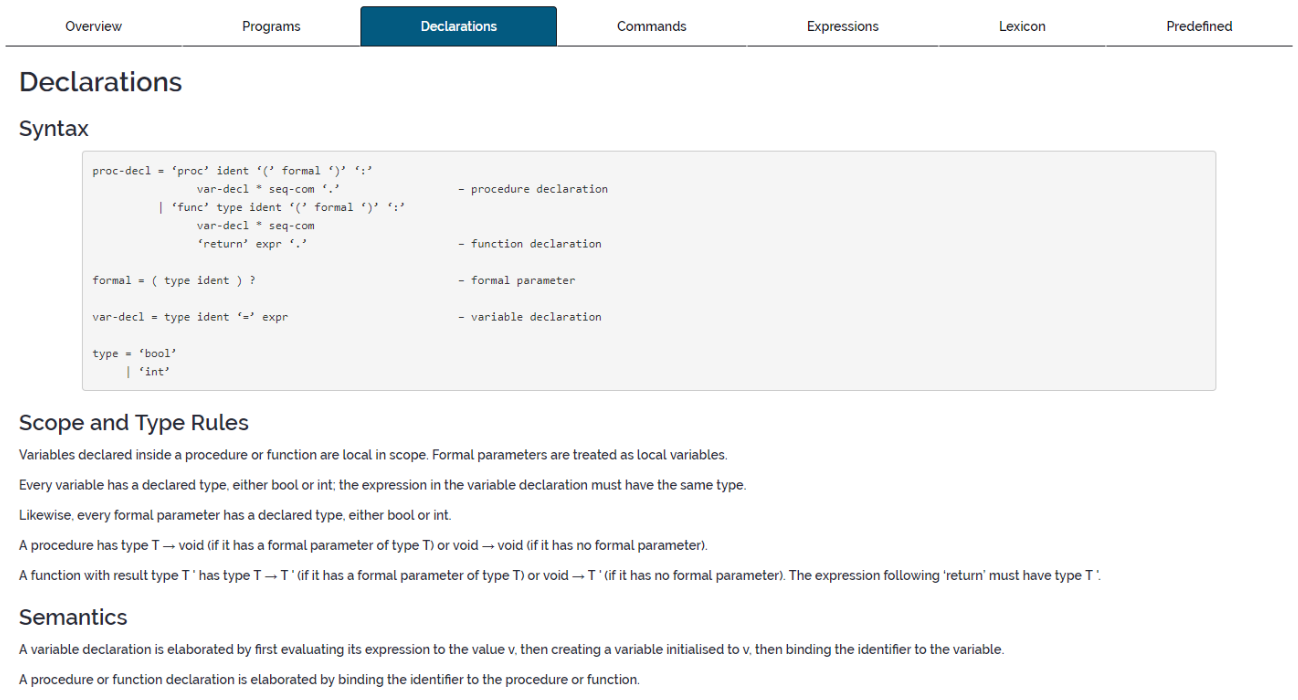
Note that this application is not a general purpose interpreter for the Fun language and does behave the way a standard REPL might. The resulting object is never executed. This does mean that any syntactically correct program can be entered and visualised, regardless of contextual errors, infinite loops, etc...

Here is an example of a Fun program:

```
bool verbose = true

func int fac(int n): # Returns 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()
  .
.
```

This program declares a global variable verbose, a function fac, and a procedure main. The function fac has a formal parameter, and ends by returning a result. A procedure may also have aformal parameter (although procedure main does not), but does not return a result. Both functions and procedures may declare local variables.

Figure 7.9a: Implementation of the Fun specification



Declarations

Syntax

```
proc-decl = ‘proc’ ident ‘(’ formal ‘)’ ‘:’
                var-decl * seq-com ‘.’                      - procedure declaration
          | ‘func’ type ident ‘(’ formal ‘)’ ‘:’
                var-decl * seq-com
                ‘return’ expr ‘.’                           - function declaration

formal = ( type ident ) ?                                   - formal parameter

var-decl = type ident ‘=’ expr                              - variable declaration

type = ‘bool’
     | ‘int’
```

Scope and Type Rules

Variables declared inside a procedure or function are local in scope. Formal parameters are treated as local variables.

Every variable has a declared type, either bool or int; the expression in the variable declaration must have the same type.

Likewise, every formal parameter has a declared type, either bool or int.

A procedure has type T → void (if it has a formal parameter of type T) or void → void (if it has no formal parameter).

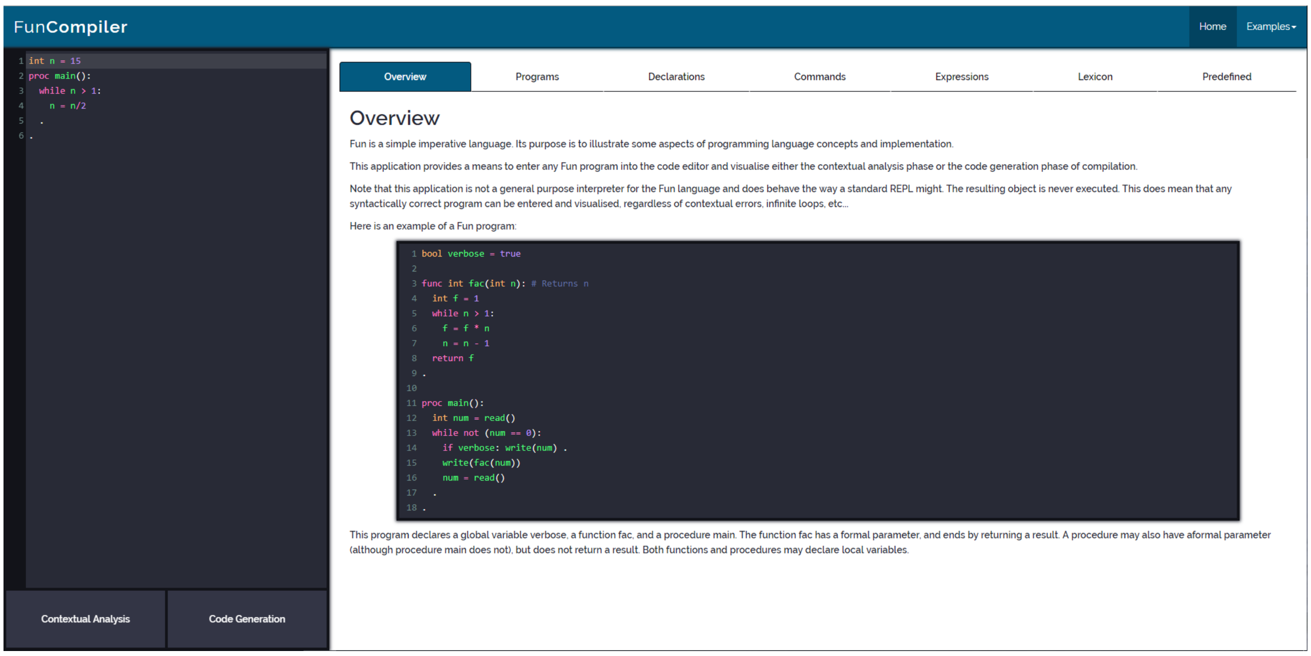
A function with result type T ' has type T → T ' (if it has a formal parameter of type T) or void → T ' (if it has no formal parameter). The expression following ‘return’ must have type T '.

Semantics

A variable declaration is elaborated by first evaluating its expression to the value v, then creating a variable initialised to v, then binding the identifier to the variable.

A procedure or function declaration is elaborated by binding the identifier to the procedure or function.

Figure 7.9b: Implementation of the Fun specification



```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

Overview

Fun is a simple imperative language. Its purpose is to illustrate some aspects of programming language concepts and implementation.

This application provides a means to enter any Fun program into the code editor and visualise either the contextual analysis phase or the code generation phase of compilation.

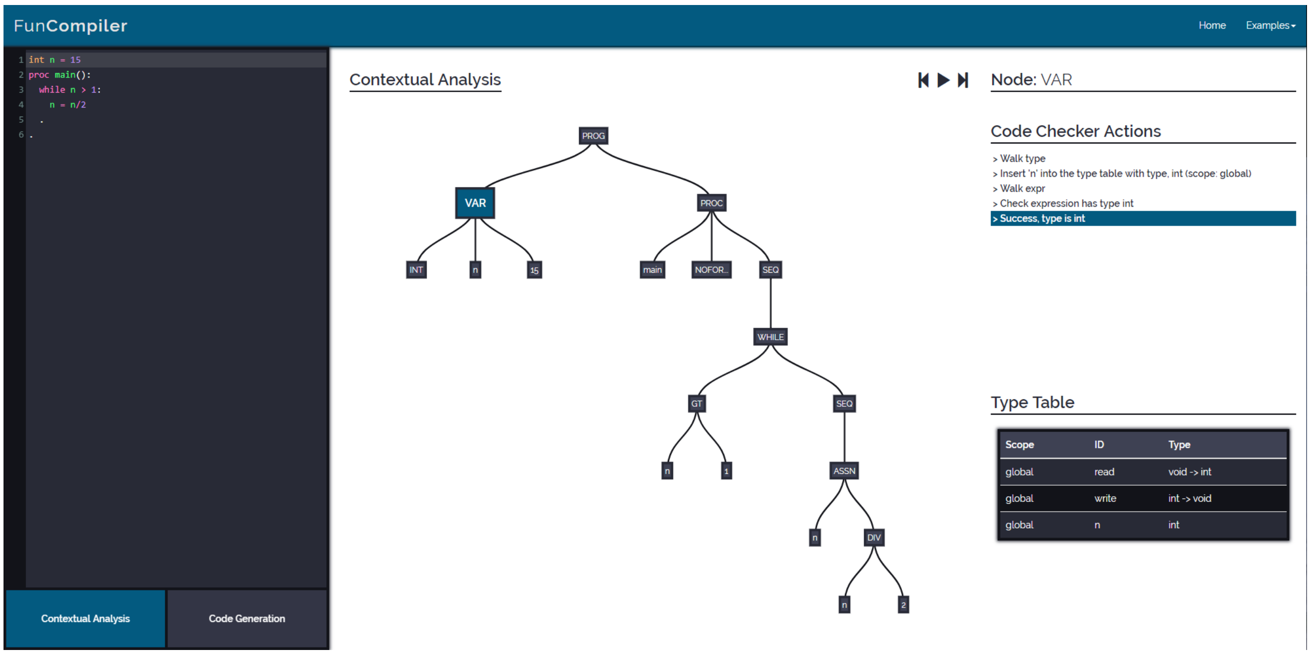
Note that this application is not a general purpose interpreter for the Fun language and does behave the way a standard REPL might. The resulting object is never executed. This does mean that any syntactically correct program can be entered and visualised, regardless of contextual errors, infinite loops, etc...

Here is an example of a Fun program:

```
bool verbose = true

func int fac(int n): # Returns 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()
  .
.
```

This program declares a global variable verbose, a function fac, and a procedure main. The function fac has a formal parameter, and ends by returning a result. A procedure may also have aformal parameter (although procedure main does not), but does not return a result. Both functions and procedures may declare local variables.

Figure 7.10a: The complete implementation of the FunCompiler - Landing page



```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

Node: VAR

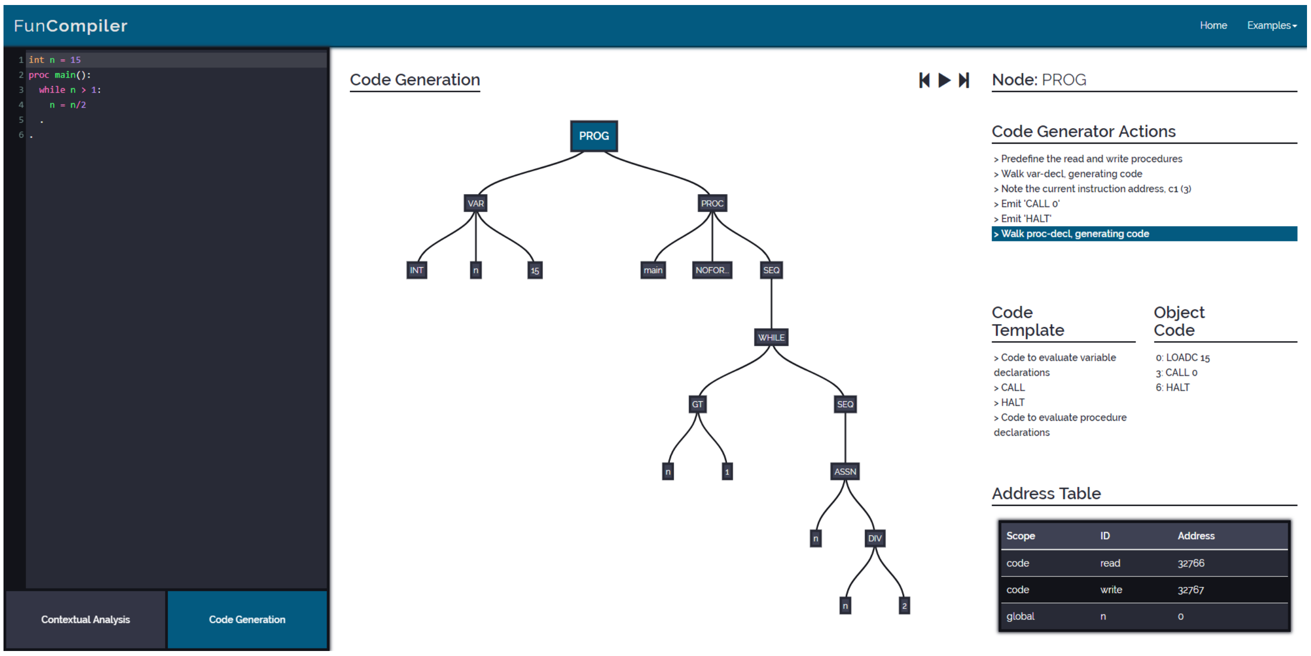
Code Checker Actions

> Walk type
> Insert 'n' into the type table with type, int (scope: global)
> Walk expr
> Check expression has type int
> Success, type is int

Type Table

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

Figure 7.10b: The complete implementation of the FunCompiler - Contextual analysis animation page



```
int n = 15
proc main():
  while n > 1:
    n = n/2
  .
.
```

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

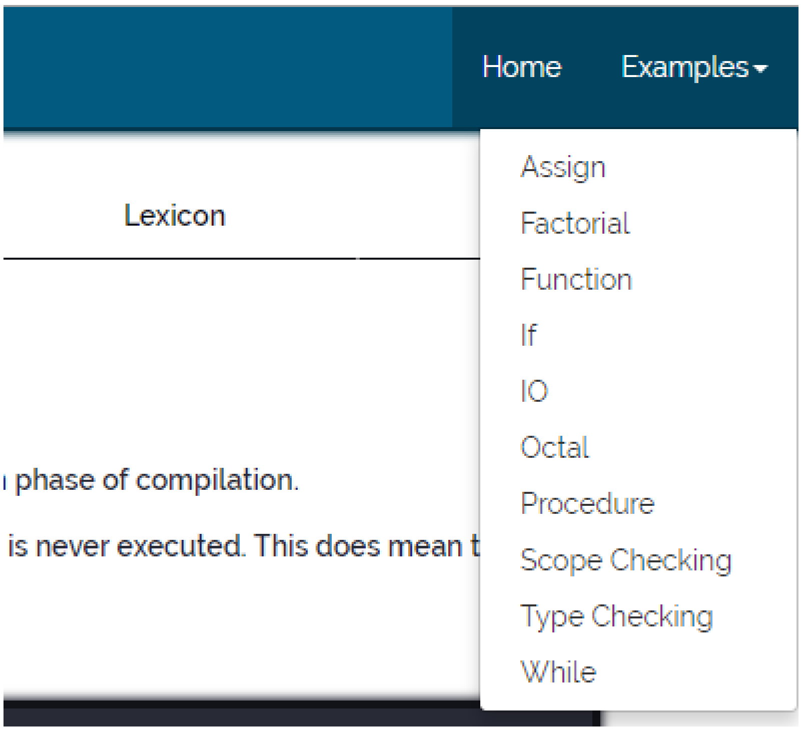
Object Code

0: LOADC 15
3: CALL 0
6: HALT

Address Table

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

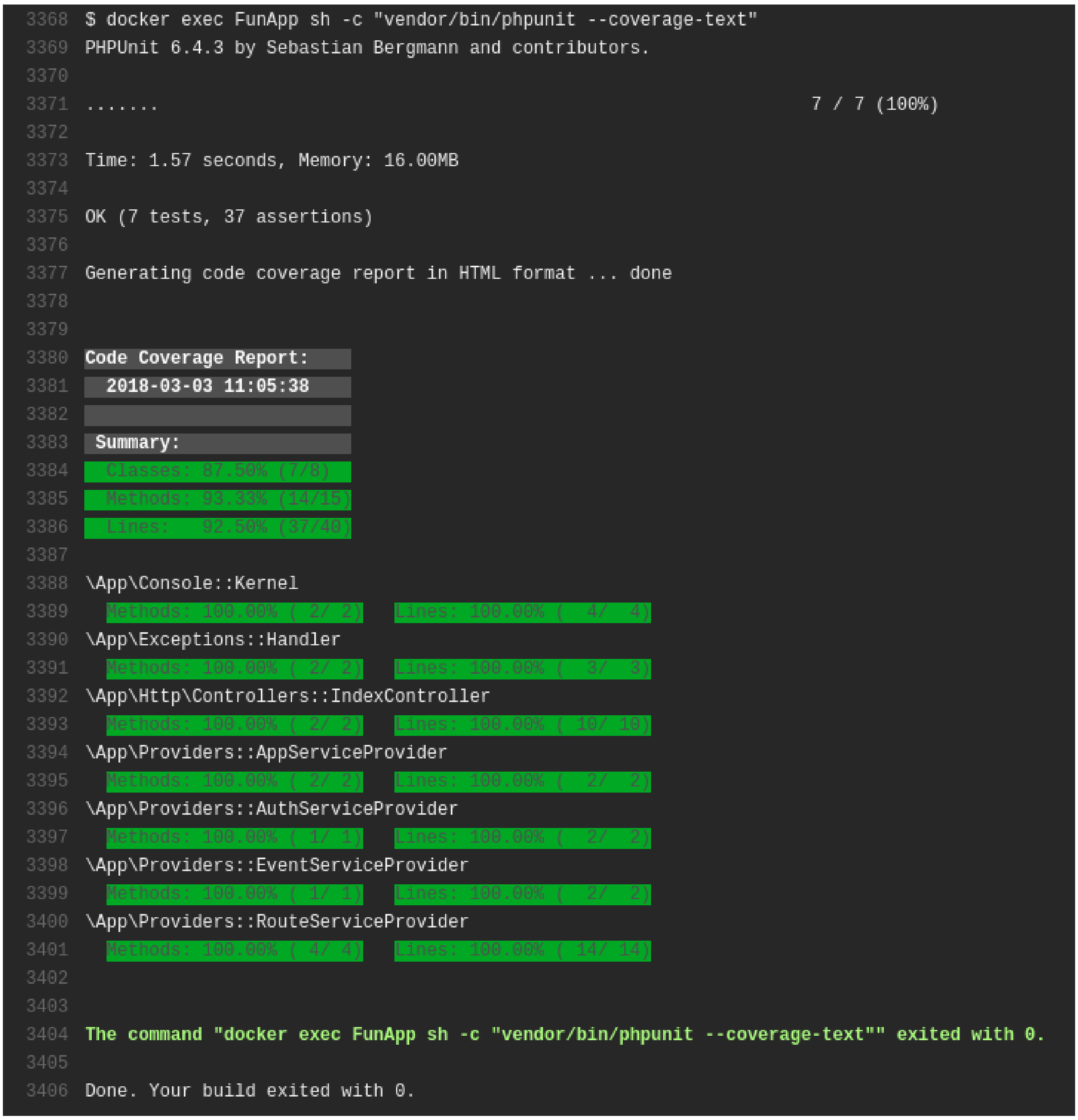
Figure 7.10c: The complete implementation of the FunCompiler - Code generation animation page



ı phase of compilation.

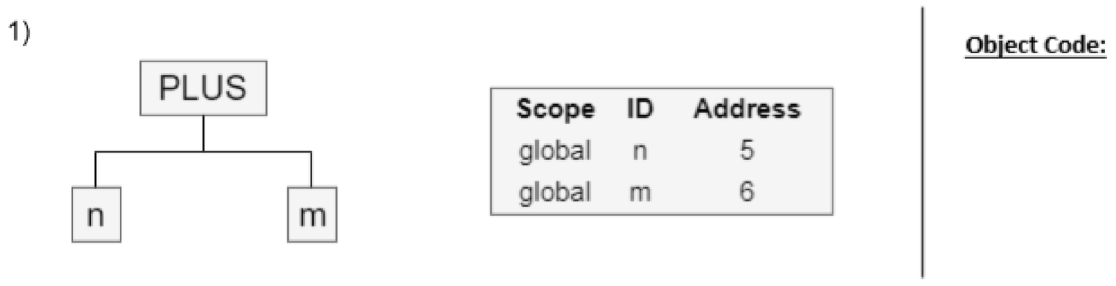
is never executed. This does mean t

Figure 7.11: Drop-down menu of example Fun programs



```
$ docker exec FunApp sh -c "vendor/bin/phpunit --coverage-text"
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.......                                                             7 / 7 (100%)

Time: 1.57 seconds, Memory: 16.00MB

OK (7 tests, 37 assertions)

Generating code coverage report in HTML format ... done


Code Coverage Report:
  2018-03-03 11:05:38

 Summary:
  Classes: 87.50% (7/8)
  Methods: 93.33% (14/15)
  Lines:   92.50% (37/40)

\App\Console::Kernel
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  4/  4)
\App\Exceptions::Handler
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  3/  3)
\App\Http\Controllers::IndexController
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% ( 10/ 10)
\App\Providers::AppServiceProvider
  Methods: 100.00% ( 2/ 2)   Lines: 100.00% (  2/  2)
\App\Providers::AuthServiceProvider
  Methods: 100.00% ( 1/ 1)   Lines: 100.00% (  2/  2)
\App\Providers::EventServiceProvider
  Methods: 100.00% ( 1/ 1)   Lines: 100.00% (  2/  2)
\App\Providers::RouteServiceProvider
  Methods: 100.00% ( 4/ 4)   Lines: 100.00% ( 14/ 14)


The command "docker exec FunApp sh -c "vendor/bin/phpunit --coverage-text"" exited with 0.

Done. Your build exited with 0.
```

Figure 7.12: Successful build and test execution of the FunCompiler via Travis CI


1)

Scope	ID	Address
global	n	5
global	m	6

Object Code:

Figure 8.1: Example question taken from the evaluation quiz

13. Position of messages on screen inconsistent ☐☐☐☐☐☐☐☐☐☐ consistent

Figure 8.2: Example question taken from the questionnaire

Bibliography

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

[2] Apple. Model-View-Controller. `https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html`. [Online; accessed 10-February-2018].

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

[4] Michael D. Byrne, Richard Catrambone, and John T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Comput. Educ.*, 33(4):253–278, December 1999.

[5] John P. Chin, Virginia A. Diehl, and Kent L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, pages 213–218, New York, NY, USA, 1988. ACM.

[6] Travis CI. Travis CI - Test and Deploy with Confidence. `https://travis-ci.com/`. [Online; accessed 10-February-2018].

[7] CodeMirror. CodeMirror. `https://codemirror.net/`. [Online; accessed 10-February-2018].

[8] D3. D3 - Data Driven Documents. `https://d3js.org/`. [Online; accessed 10-February-2018].

[9] Martin Fowler. Microservices Resource Guide. `https://www.martinfowler.com/microservices/#what`, 2015. [Online; accessed 10-February-2018].

[10] Hayim Makabee. Separation of Concerns. `https://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/`, 2012. [Online; accessed 10-February-2018].

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

[12] PHPUnit. PHPUnit - The PHP Testing Framework. `https://phpunit.de/`. [Online; accessed 10-February-2018].

[13] ReplIt. Online REPL, Compiler and IDE. `https://repl.it/`. [Online; accessed 14-March-2018].

[14] Webopedia Staff. SVG - Scalable Vector Graphics. `https://www.webopedia.com/TERM/S/SVG.html`. [Online; accessed 10-February-2018].

[15] John T. Stasko, Albert N. Badre, and Clayton Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In *INTERCHI*, 1993.

[16] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, March 1987.

[17] Restful API Tutorial. What is REST. `https://restfulapi.net/`. [Online; accessed 10-February-2018].

[18] VisuAlgo. VisuAlgo - Visualising Data Structures and Algorithms Through Animation. `https://www.visualgo.net/en/sorting`. [Online; accessed 03-February-2018].

[19] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1976.

[20] Niklaus Wirth. *Theory and Techniques of Compiler Construction*. Addison-Wesley, 1996.