University of Plymouth

School of Engineering,

Computing, and Mathematics

PRCO304

Final Stage Computing Project

2019/2020

Compiler with Visualisation of Data Transformations

Callum Todd

10555310

BSc (Hons) Computer Science

Acknowledgements

My thanks to my project supervisor, Vasilios Kelefouras, and my patient family.

Abstract

Compilers are often portrayed as a complex ‘black box’ piece of software. This report outlines the aim to create a visualisation tool showcasing the data transformations required for compilation, using a functional toy compiler as the back-end.

The report discusses the project’s aims and objectives, along with looking at the legal, social, ethical, and professional issues surrounding the project. Also detailed is the method of approach taken towards the development of the solution and the project management techniques employed.

The main body of the report sees a breakdown of the solution’s development journey, along with the architectural and design decisions made. Closing the report is a review and critical evaluation of the project as a whole, along with specific retrospectives delving into the project management, technologies used, and decisions made. Finally, there is an appendices section where a user guide for the solution is given, along with other materials.

Table of Contents

Word Count

Word count: 8941

Code Repository

The developed solution's code can be found at:

https://github.com/CallumTodd7/compiler-visualization

1 Introduction

1.1 Background & Existing Solutions

1.1.1 Background

This project is built upon personal experience when learning about compilers from both a self-taught and academic point of view. Often times compilers are portrayed as being a black box; more complex than other software. When decomposed, compilers use the same fundamental development techniques that are used throughout the industry. From personal experience, there is very little material that sits between the basics and advanced compiler theory. This project aims to create a solution that sits in this middle ground and caters to visual learning principles. When researching in years prior to this project, the concept of visual learning only made rare occurrences in the abundance of resources on the topic originating from the pre-2000's. Confounding this issue was the limited resources covering this middle ground from more recent years.

Visual learning is where data, ideas, and concepts are associated with graphics and animations, allowing students to better grasp advanced and abstract topics (Janitor, 2010). The inherently abstract nature of compliers lends itself to being taught via visual learning for those which can benefit from it.

1.1.2 Existing solutions

The idea of visualising software is not new. There are a selection of tools designed to visualise code execution, as it would happen on a CPU. VisUAL is an emulator for ARM assembly aimed at being an easy entry-point into ARM development (Arif, 2015). While VisUAL does not visualise any data structures, it does provide visual debugger-like features in a way that makes it valuable for learning. Python Tutor is a web-based

interpreter for many languages, visually displaying memory blocks alongside the source code being stepped through (Python Tutor, n.d.). There are many more tools like these available that visualise how code gets executed, however there are few tools that lean towards compilation.

Visual YACC and PAVT are tools among a handful that visualise the front-end of a compiler, i.e. the lexer and parser. Visual YACC is a visualiser built on top of the popular parser generator YACC (White, 1999), and PAVT (Parsing Algorithms Visualization Tool) is a piece of software that is capable of visualising many different parsing algorithms (Sangal, 2018).

Despite there being front-end compiler visualisation tools, there are few, if any, tools centred around visualising the entire end-to-end compilation journey. This would add value as it would allow the user to see their source code transformed at every step in the process for one particular compilation method.

1.2 Aims, Objectives and Deliverables

1.2.1 Project Aims

The aim of the project is to create a visualisation tool that shows the internal workings of a compiler. The solution will have two core areas: a functional compiler, and a visualisation front end, which will produce a graphical and animated representation of the source code being transformed into the outputted assembly. The end user will be able to write source code in a given programming language, pass it to the solution which will visualise the stages of compilation, and then receive an outputted assembly file which will assemble to a valid executable.

1.2.2 Project Objectives

The project's objectives are as follows:

- Conduct user research to validate visualisation concepts within the first 6 weeks of the project[1].
- Develop a custom toy compiler to back the visualisation. The core functionality of this must be completed within the first 5 sprints[2].
- Develop a programmatic visualisation of a compilation flow. This must be completed after the compiler's core functionality has been developed and before the end of the project.

1.2.2.1 Solution Minimum Requirements

The project solution will be able to:

- run on macOS (discussed in the Tools & Technologies chapter)
- compile a source code text file into an assembly text file
  - The source code programming language will be a custom C style language (details can be found in the User Guide appendix)
  - The output will be targeted for x86-64 assembly (NASM was chosen as the target assembler), creating Linux/Ubuntu ELF binaries
- perform some limited semantic analysis of the end user's code (i.e. detect if a variable is being used before definition)
- display end-to-end visualisations of the compilation process on screen
  - This will not show every single atomic step, however enough detail should be provided for someone to be able to follow along, or allow someone with knowledge to talk over the visualisation.

The proposed compiler's programming language will feature:

---

[1] The visualisation concepts will need to be finalised before development next can begin on the visualisation front-end, which will occur just after the 6 week mark.

[2] Sprint 6 begins work on adding compiler-backed visualisations to the solution.

- multiple scopes (i.e. file scope, procedure scope, block scope)
- procedures
    - with support for parameters and return values
    - compatible with the C calling convention[3]
- variable declarations and assignments
- `if` statements
- `while` statements
- arithmetic operators (i.e. addition, subtraction, multiplication, division)
    - Supporting integer literals and variables

1.2.2.2 Solution Stretch Goals

Time permitting, as optional goals, the project solution might be able to:

- allow the user to scrub through the visualisation, and pause and alter playback speed **[Partially achieved[4]]**
- export a video of the visualisation
- provide detailed user facing errors occurred during compilation through semantic analysis

Time permitting, the compiler's programming language might feature:

- string literals **[Achieved]**
- more complex data types (i.e. arrays, strings, structs)

---

[3] This is so procedures from the C standard library can be used, thus saving time. `printf` and `puts` can be used without having to bundle a runtime library with a compatible calling convention.

[4] The ability to skip forward through the visualisation was introduced, however there is no scrub bar or support for skipping backwards through time.

- out of order definition (i.e. defining a procedure after it is called in a lower sibling scope)
- expression logical operators (i.e. AND/&&, OR/||)

1.2.2.3 Planned Solution Limitations

Because the project's scope could potentially be perceived as higher than it is, it is equally important to define the scope and limitations of the project in addition to its goals. Therefore, the solution does not aim to:

- be performance efficient, or produce efficient output executables. Providing insight to people is the goal of the project, not producing efficient binaries. Therefore, it is more important for the solution to be clear and simple to understand, over optimisations improving performance while clouding core concepts.
- be extensive. The outputted assembly will only be using a limited instruction set. For example, while there are many instructions capable of summing two registers in the x86 instruction set, the `ADD` instruction will be used solely throughout. This is to save time and simplify the work required, making it viable for a single developer to create it in the allotted time. Again, a fully featured compiler is not an aim of this project.
- be usable for real-world programming projects. The solution's compiler will be a 'toy compiler' designed to show off how compilers work.
- be a de facto guide to how compilers work. It will show how one specific compilation method works, however there are many other methods and techniques to compile software that will not be represented here. The goal of the project is to provide an opening for others' learning, not to be the full solution.

1.2.3 Deliverables

The following lists the high-level deliverables that will have be produced for the project:

- A compiler visualisation tool; the solution of this project

- A user guide for:
    - the application/solution
    - the programming language that the compiler understands
- This project report

2 Legal, Social, Ethical, and Professional Issues

2.1 Legal

2.1.1 Licences and Intellectual Property

All third-party code and assets are being used in accordance with their licences. All immediate dependencies have been listed in the solution's README.md file and their licence appended to the solution's LICENCE file for clarity.

There are four immediate third-party libraries/assets included with the solutions source code:

- Love2d, a game engine for Lua. This is under the zlib licence (Love2d, n.d.) which permits redistribution and modification for any purpose, so long as it is clearly labelled as such (Opensource.org, n.d.). This library has been modified for its use within the solution to remove many game related features and, significantly, Lua support. The solution only makes use of its C++ backed graphics APIs. This has been clearly stated in the solution's README.md file.
- SDL2, an API that abstract many low-level hardware functions. This is also licensed under zlib (Simple Directmedia Layer, 2020), permitting redistribution and modification for any purpose, so long as it is clearly labelled (Opensource.org, n.d.). For this project, the library's source code has simply been bundled in the code repository and is built 'as is' and statically linked to the solution's target executable.
- FreeType, a font library. This is licensed under the FreeType Project Licence (FreeType, 2018), permitting redistribution with or without modification with the condition that credit is given is the end user documentation for the product (GNU

Savannah Git Hosting, 2006). This clause has been observed and a statement exists in the solution's user guide; see Appendix 1. Similarly, the library's source code has simply been bundled in the code repository and is built 'as is' and statically linked to the solution's target executable.

- Source Code Pro, a monospace font. The font is licensed under the SIL Open Font Licence (Adobe Fonts, 2019), permitting redistribution and modification (Opensource.org, n.d.). This project bundles the unmodified TTF font after it has been converted to a C file containing an array of bytes, exactly matching the original TTF. Because the licence permits modification, this conversion is permitted regardless of whether it would be classified as modification.

The solution has been licensed under Creative Commons Attribution 4.0, and the source code is available publicly on GitHub.

2.1.2 Laws

The solution application processes everything locally on device and does not use network access or store any ancillary data with use (i.e. metadata). As a result, there are no data protection implications.

2.2 Social, Ethical, and Professional

Due to the goal of producing an educational tool, a duty of care to provide accurate information to the end user must exist. Due to the fundamental construction of the solution, where the visualisation is directly driven from a functional compiler as the back-end, there is not significant concern of misleading information being portrayed. As stated in the project's objectives, the solution does not aim to serve as a complete framework to an individual's education on the topic, instead only as an introduction to go alongside either narration from an expert or some other adjacent resource.

3 Method of Approach

3.1 Project Structure

At the start of the project, an initial high-level plan was created to act as a support structure, for both the development of the solution and the project as a whole. The plan breaks the project down into four main phases:

1. Planning
2. Compiler Development
3. Visualisation Development
4. Project Analysis and Report

These phases roughly map to the sprint breakdown detailed in the Development chapter.

The Planning phase was vital to ensure that the future phases in the project were performed correctly. The goal being to identify tools and technologies best suited to the project, along with architecting the solution at a high level, ensuring a smoother development cycle.

The development of the compiler and the front-end visualisation was split into two distinct phases because of the widely different problems they are solving, and the technical solutions needed to achieve them. In addition, the font-end visualisation is heavily dependent on the compiler being completed to a *minimum viable product* before work could begin.

Ending the project would be the completion of this written report.

3.2 Agile

The agile methodology centres around continuously delivering functional software at the end of each development period, known as a sprint. The idea being that continual iteration and improvement leading towards the completed version, and beyond, allows for development teams to respond quickly to change (Atlassian, n.d.). This is opposed to the waterfall approach where the entire solution is integrated late in the development process, and is thus prone to delays.

Kanban was chosen to be the exact agile methodology to be followed for this project due to its simplistic approach, lending well to single developer teams.

Integral to the foundation of being able to respond quickly to change is the idea of just in time planning. To aid in this, spike tasks may be added to the backlog, indicating an investigation work item on a particular topic. Spike tasks are useful when an element required for the design of the solution is unknown, as found with the graphics library to be used for the visualisation, among other areas in the solution.

To aid in estimating how much work can be completed in a given sprint, story point estimates are commonly used to determine the size of each task in the backlog. For this project story points in the Fibonacci sequence were used when estimating; a lower point value being a quicker task, while a higher point value taking a longer duration. At the end of each sprint, story points for future tasks in the backlog are re-estimated to ensure they are as accurate as can be.

3.3 Testing

For the compiler half of the solution, regression testing is performed using a collection of input files. A bash script executes the program with each source file in-turn and display the process status codes if any runs fail. The test script is automatically ran

regularly via custom scripts on the author's development machine[1]. While an incredibly simple setup, it provides a level assurance that the core functionality of the application has not regressed from when the test was written.

Manual testing is performed for the visualisation half of the solution. Due to the extremely linear flow the program takes and the lack of external factors at play (i.e. limited user input), automated testing would be of limited benefit beyond what manual testing can achieve. Non-extensive manual testing is inherently actioned during development, reducing the number of extensive manual testing runs required to spot issues. In all, for this specific part of the application, it has most likely been more time efficient to perform regular manual tests, over writing automated tests to reduce this time.

3.4 Tools & Technologies

3.4.1 C++

C++ was quickly identified to be the programming language for the solution due to having the right balance of low-level features, making data mutations easier, and higher-level features, allowing for a faster development time. The author's existing experience with the language also applied a significant weighting to the choice.

Not chosen, but considered, was Python for its ease of use in rapid development and the extent of libraries available. In the end Python was not chosen due to the author's preference of non-*duck typed*[2] languages.

---

[1] If multiple people were interacting with the codebase, or there were stakeholders involved with the development, more formal/accessible automation would have been used. Most likely in the form of GitHub Actions or a Jenkins server.

[2] A *duck typed* language is, for all intents and purposes of this discussion, the opposite of a statically typed language.

3.4.2 CMake

After C++ was chosen, CMake was a natural step with it being a widely used build system for C and C++ projects (CMake, n.d.). This was particularly important as the author's development machine runs macOS. In addition, any third-party libraries being used for the solution would be able to integrate with CMake.

3.4.3 Love2d

For the visualisation half of the solution, a graphics library would be needed to make development of a graphical front-end feasible. While the option of using a low-level rendering engine such as OpenGL or Vulkan existed, this was quickly dismissed as being too low level for the solution's needs. The game engine library Magnum was originally chosen after the initial research spike, however when it came to integrate with Magnum the library was quickly abandoned. This was due to the scale of the library mandating a larger learning curve and initially expected. This would have required a significantly greater amount of time to learn than was available.

Looking for a simpler graphics library, the Lua based game engine Love2d was considered due to the author's past experience using it. Despite being a library for a different programming language, Love2d is backed by a C++ and is completely open source (Love2d, 2020). It is based on a suite of industry standard libraries that the author also has past experience with, such as SDL2 and FreeType (Love2d, 2020), meaning there wouldn't be a large learning curve for the author when reading its source code. Theoretically, due to its C++ underpinnings, the graphics engine could be tapped into from C++ without the need to use Lua at all. After some investigation to the feasibility, this path was chosen and a copy of the library was modified to remove Lua support and many other game related modules that would not be used.

3.4.4 macOS

Due to the author having primary access to macOS platforms, the solution has been developed on and for macOS. Despite this, intentionally only cross platform

components have been used to ensure only relatively minor tweaks are needed to get the solution working on a platform other than macOS.

3.4.5 Docker (Ubuntu)

A Linux based OS, specifically focusing on Ubuntu, is required to execute the compiler's outputted assembly code (NASM, 2018). This is because the decision was made for the compiler to generate x86-64 instructions targeting the Executable and Linkable Format (ELF) and the Linux kernel (Tool Interface Standard, 1995). In order to make development and running more convenient, Docker is used to setup a virtual Ubuntu environment on the host machine where the compiler can be executed (Docker, 2020). A `Dockerfile` is checked in to the version control and Docker's use documented in the User Guide appendix.

Alternatively, it is still possible for the user to transfer the outputted assembly to a native Ubuntu install and execute the program from there.

4 Project Management

4.1 Trello

The development of the solution has been managed under an agile Kanban workflow. To facilitate this, an online project management tool was used. Trello offers a number of columns for cards/tasks to be placed in and moved between, the foundation of Kanban style workflows.

For this project, the columns on the Trello board are as follows: Backlog, Blocked/Waiting, In Progress, and Done 🎉. Each card in the backlog represents a body of work to be completed, along with a story point estimate of the work[1], and several labels for easy identification of the work type (i.e. development, design, user research, stretch goal, etc).

Pinned to the top of the Backlog and In Progress columns is the Definition of Ready[2] and the Definition of Done respectively. These serve as guidelines that must be met before the card can be moved along to the next column in the swim lane. Having the work tickets conform to a set standard before proceeding forward ensures a greater level of quality and helps prevent mistakes from slipping through.

The Definition of Ready was defined as:

- Decomposed to individual tasks
- Decomposed to take less than one week
- Sufficient detail of what is required

---

[1] In the Fibonacci scale

[2] As in *ready to be worked on*

And the Definition of Done defined as:

- Ticket task complete, or if not in full another ticket for the remaining work has been created
- Test programs updated/new ones added
- Documentation updated

A screenshot of the Trello board during each sprint can be found in the Project Management – Trello Progression appendix.

4.2 GitHub

Version control is vital to software development projects as it allows for revisions to be stored of the codebase, safeguarding against potential loss of data by mistake or corruption, and allowing for traceability of the code (Atlassian, n.d.). Git is a widely used industry standard Version Control System and has been chosen for this project.

GitHub is a popular Git repository hosting company. The use of a remote repository acts as an offsite backup for the solution's code, along with allowing for strong collaboration features allowing the code base to be worked on from multiple locations by different collaborators. For this project, the solution was developed by the author on several machines, with the remote repository stored on GitHub streamlining the process of keep code in sync across devices.

4.3 Show and Tells

Show and tells, or sprint reviews, are regular meetings where the past sprint's work is demonstrated to various stakeholders (GOV.UK, 2019). These are extremely useful in team environments and those working in organisations, but also have use in the context of this project too. Regular supervisor meetings acted as show and tell sessions, demonstrating and discussing the past sprints activities.

5 Architecture and Design

The solution has two distinct areas that are bridged together: the compiler; and the front-end visualisation. The compiler runs on a background thread allowing it to execute without blocking the graphics running on the main thread. Throughout the compilation process, the compiler will post events about the current execution state. The visualisation system polls for a new event once the previous event has been visualised and animated.

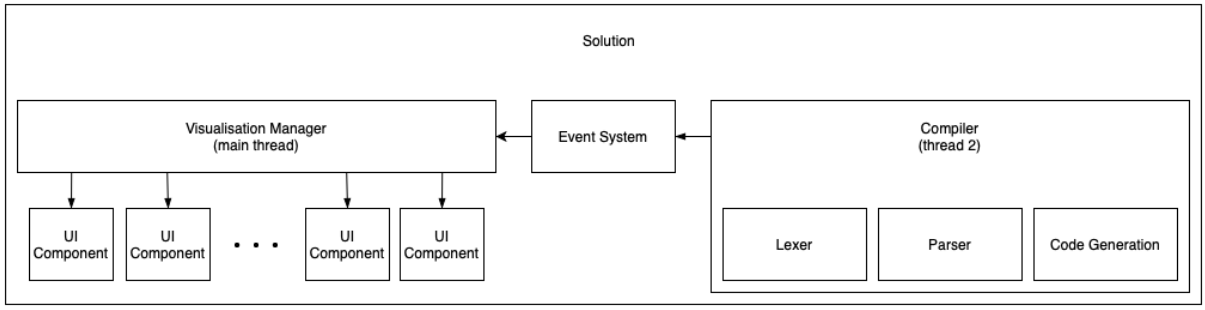

Figure 1 High-level system overview

There is also a command line flag to enable a non-UI mode where the visualisation system is disabled, and the compiler's posted events are disregarded.

5.1 Compiler

The compilation process used in the compiler has three main phases in sequence: lexical analysis, parsing, and code generation. The source code is tokenised using lexical analysis before being parsed into an intermediate representation. An intermediate representation is beneficial to remove the lexical meaning imbued in the source code, striping the information back to pure semantic meaning. This allows further phases (i.e. code generation) to not be concerned with the programming

language's format. Finally, the immediate representation is traversed to generate an output assembly code file.

In more complex/fully featured compilers, semantic analysis can often be performed in its own phase after the intermediate representation has been built (i.e. parsing) and before code generation occurs. In a strongly typed language, or a language with type influence, there is often additionally a whole phase specifically for typing. This separation of responsibilities into different phases is beneficial for decomposing large problems into smaller manageable code areas that can benefit from other parts of the program being abstracted away. Due to the design of this compiler not being of sizeable complexity, there is no need for this level of separation.

5.1.1 Lexer

The lexing phase is responsible for taking the stream of characters from an input source file and translating them into a sequence of tokens for the parser to interpret.

A token represents a language keyword or feature, or an integer or string literal. As well as the type of token, each token can also hold an optional value in the case of an integer or string literal. Additionally, tokens store contextual information about where they were generated from in the source code. This extra context can be used to generate more precise error messages, among other uses.

The method used to perform the lexical analysis involves stepping through each character from the input file character stream. Looking at the first character in the stream, it is categorised into one of the following groups: whitespace or comment start characters; digit characters (0-9); alphabetic character (a-Z); or *other* chapters, typically punctuation/operators. For each character group there is a subroutine that will supervise the creation of a single token. Each subroutine will read forward from the input character stream until a character is encounter that the subroutine cannot process. At this point a token will be generated from the read data and is added to the output sequence of tokens. This cycle of categorisation and handling will repeat until the end of the file is reached.

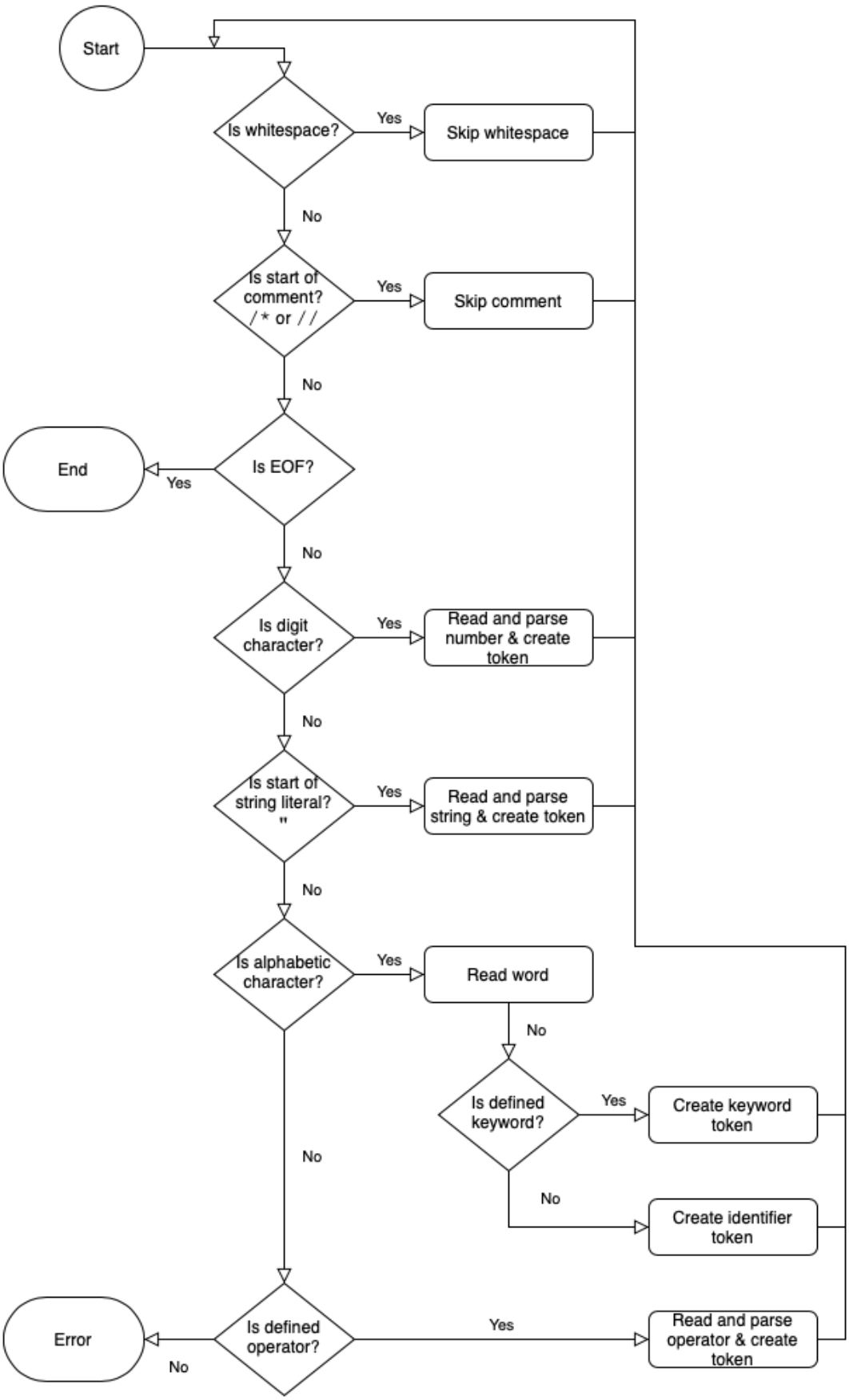

Figure 2 Flowchart of lexer decision flow

5.1.2 Parser

The parser is responsible transforming the lexer's sequence of tokens into an intermediate representation in the form of an AST. The parser expects the token stream to conform the programming language's grammar. If the input does not conform, an appropriate error will be outputted and the compiler's execution halted. A grammar defines the rules of a programming language. The grammar of the compiler's programming language is defined using an Extended Backus–Naur Form (EBNF) grammar and can be found in the Language EBNF Grammar appendix.

While there are many methods for parsing programming languages, Recursive Decent Parsing was chosen for being simple to implement as it takes advantage of recursion features built into the language the compiler was written in[1]. It also has the added benefit of mapping closely to an EBNF grammar; one grammar rule to one subroutine in the parser.

The method works by there being at least one subroutine for each node type possible in the AST[2] to handle the generation of that node. When a subroutine is running, it checks the current token for expected token types and recursively calls other subroutines to generate child nodes where required. Below is a snipped of code, modified for brevity and clarity, showcasing what the subroutine to generate a *while statement* AST node looks like:

```
AstWhile* Parser::parseWhile() {
  auto node = new AstWhile();
  expect(Token::Type::KEYWORD_WHILE);
  expect(Token::Type::PAREN_OPEN);
  node->conditional = parseExpression();
  expect(Token::Type::PAREN_CLOSE);
  node->body = parseBlock();
  return node;
}
```

[1] In this case C++, however recursion features are extremely common across languages.

[2] Examples include: integer literals, binary operations, variable assignments, statement blocks, if statements, procedure declarations, etc

5.1.3 Code Generation

The code generation phase is responsible for walking over the AST intermediate representation and outputting valid x86 instruction code to a file.

In order to output a series of instructions that can be assembled into a valid executable, the contents of each register must be known at each stage in the user program. Obviously since this is not an interpreter the exact data value will not be known, but what variable will be in what register at what moment is important to determining the instructions to output.

To achieve this, each node in the AST is given a unique value that represents a potential value or intermediate value. A map of registers to these unique values is stored in memory and can be used to determine if a register contains a value and what that value may represent. Every time an instruction is outputted that will move data between registers, the register map is updated to reflect the changes.

The list of instructions is determined by walking over the AST in a recursive manor, with each subroutine responsible for outputting instruction code for corresponding AST node type.

5.2 Bridge/Event System

If the compiler were to run on the main thread, that is also housing the visualisation graphics, there would be the potential for frame stuttering or window hangs to occur while the visualisation waits on a portion of the compiler to finish executing. In order to counter this, two threads are used: one for the compiler, and one for the graphics. An event system bridges the two, allowing for events to be posted from the compiler thread and received on the main graphics thread.

For the compiler to post an event containing relevant data about the action just performed, a function is called that stores the event on a queue to be later picked up. It also has the ability to block the compiler's thread until the main graphics thread requests for it to continue executing. This is necessary to keep the, typically faster, compiler in sync with the visualisation.

On the front-end side, a function can be called to poll for a new event from the compiler. This function returns the next event popped off the queue. If there are no events waiting, the function will return with nothing, instead of blocking the main thread, to keep the graphical window from hanging. When no data is returned, the front-end will poll for new data on the next update cycle, effectively creating a spin-lock until new a event emerges[3].

Since multiple threads are sharing data, mutexes are utilised to keep the event system is thread safe. In this implementation, mutexes are used to ensure read and write operations from each thread do not occur on the central event queue at the same time.

5.3 Front-end Visualisation

So that the compiler's process may be visualised to the user, a rendering loop is created, rendering text and coloured shapes to a buffer and then presenting the graphics to a window on the user's screen at the end of the cycle. The structure of the visualisation code is based on a series of components specialising in rendering a single visual element with the given data.

[3] There is a small delay of 0.001 seconds in between each update cycle, so CPU is not being completely maxed out.

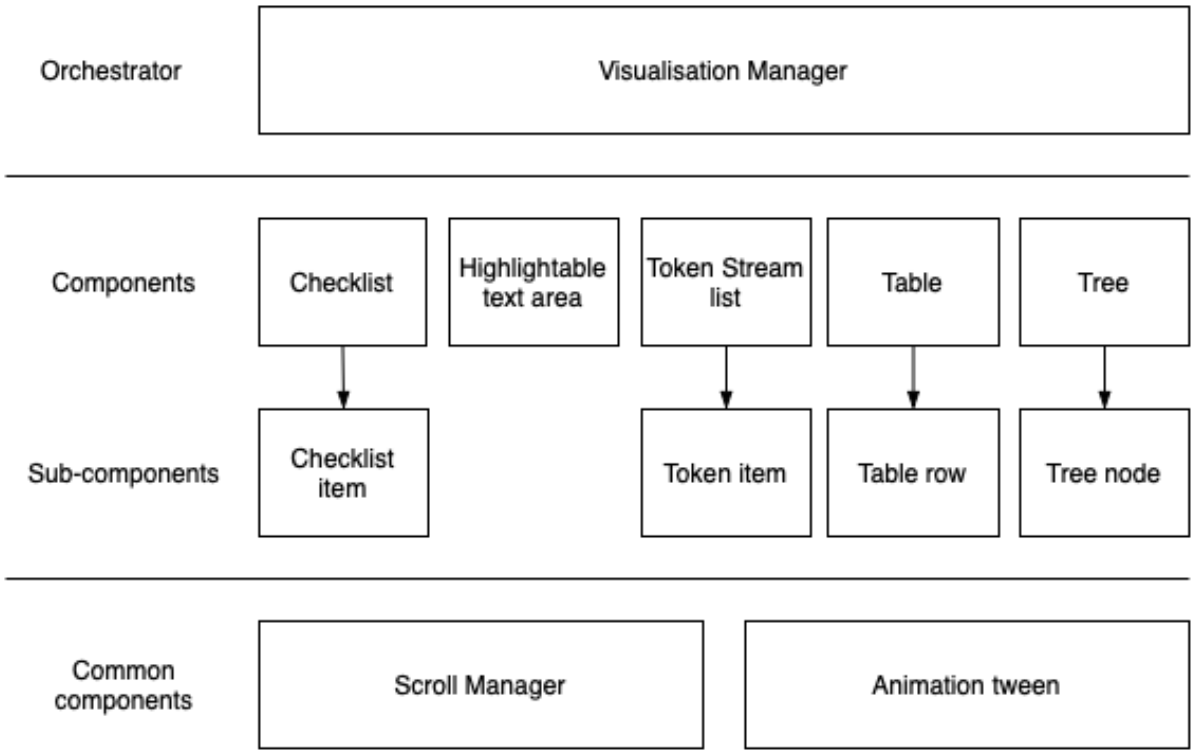

Figure 3 UI components

Each component implements three standard functions that can be called by the orchestrating class: `update`, `draw`, and `hasActiveAnimations`. The `update` function, called once per frame[4], is responsible for changing the position of any drawn shapes that are currently being animated. The `draw` function, again called once per frame, uses the global graphics context to render various shapes and text to the render buffer. Finally, the `hasActiveAnimations` function returns a boolean as true if the component houses any animations that are currently in flight. This is function is used to determine if all animations on the screen are complete, and thus the visualisation is ready to move on to the next step, polling the event system for new data from the compiler.

[4] with a time delta since the last frame cycle

5.3.1 Love2d

A graphics library was used to abstract away implementation details of the solution that does not pertain directly to meeting the aims. As previously mentioned, no off-the-shelf graphics libraries for C++[5] offered enough high-level abstraction in a simplistic manor, which was required for this project. As a result, focus turned to using a 2d game engine library written for Lua, due to the author's past experience using it. This was a viable option because it was discovered that the Love2d library is open source and backed by C++ code (Love2d, 2020), making it possible to tap into 'private' APIs directly from C++.

Since Love2d is a full game engine including subsystems for graphics, audio, physics, events, and more, it has much greater functionality than required. A significant portion of the engine's source code and its dependencies are for integrating with the Lua language. Since only the graphics and windowing [6] subsystems are needed, modifications were made to a copy of Love2d's source code to remove all irrelevant files and modify the remaining files so that the library continues to work with some parts missing.

5.3.2 Design

Gource, the Git commit visualisation tool (Gource, n.d.), was a heavy inspiration to the initial idea of this project, and as such carries a heavy design influence. The visual design of the solution is centred around the idea of using simplistic shapes to form the elements on screen, keeping visual clutter to a minimum. In order to show links between different elements on screen at the same time, colours are used to code key areas as having joint meaning. This can be seen best in the lexing phase where the currently observed characters in the source code are highlighted in the same colour as the current phrase being tokenised in the middle column on the screen.

---

5 That were discovered during research

6 The windowing subsystem allows graphics to be displayed in an operating system window.

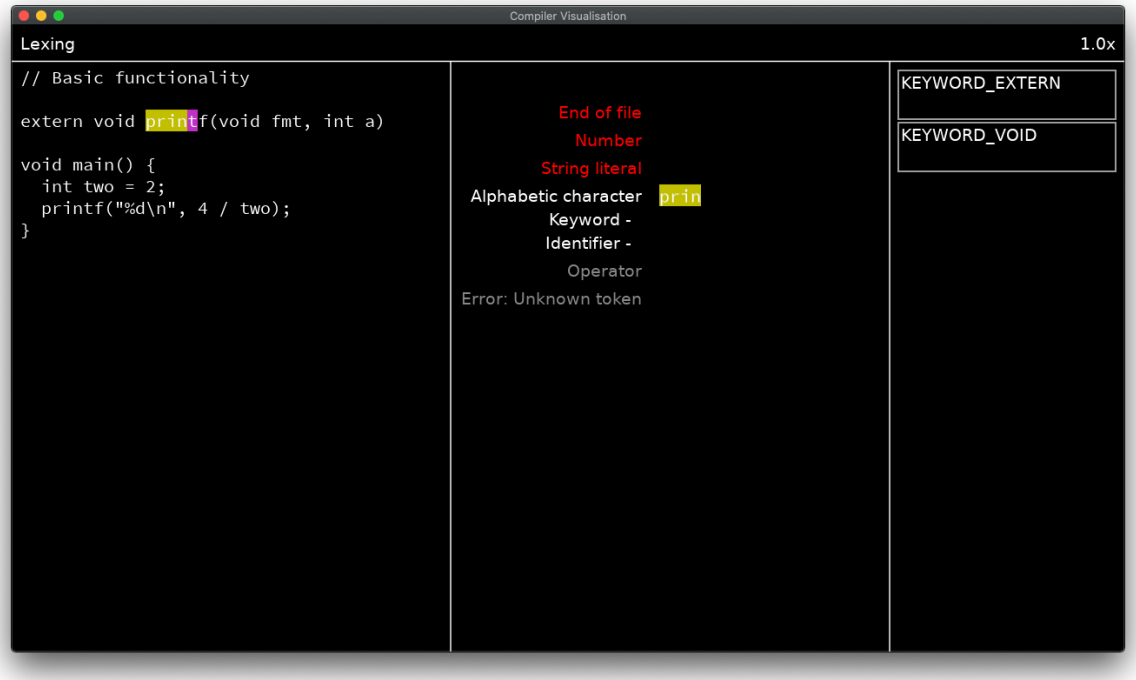

Figure 4 Screenshot of the solution's lexing phase, showcasing the colour coded highlighting

6 Development

6.1 Sprint 1 - Architecture and planning

The initial sprint focused around planning out the scope of the work required and preparing for development of the solution to start. After identifying the upcoming tasks for project, story point estimates were applied and inputted into Trello. Throughout the development processes these tasks were altered as their work became clearer, along with new tasks being added once being identified as required.

The design for the visualisation flow and aesthetic was sketched out to a conceptual level at the start of the project, acting as a guiding force through the development process. This initial design has the intention that it will be further refined with user research, which will be conducted in future sprints after ethic approval has been granted. These sketches can be seen below.

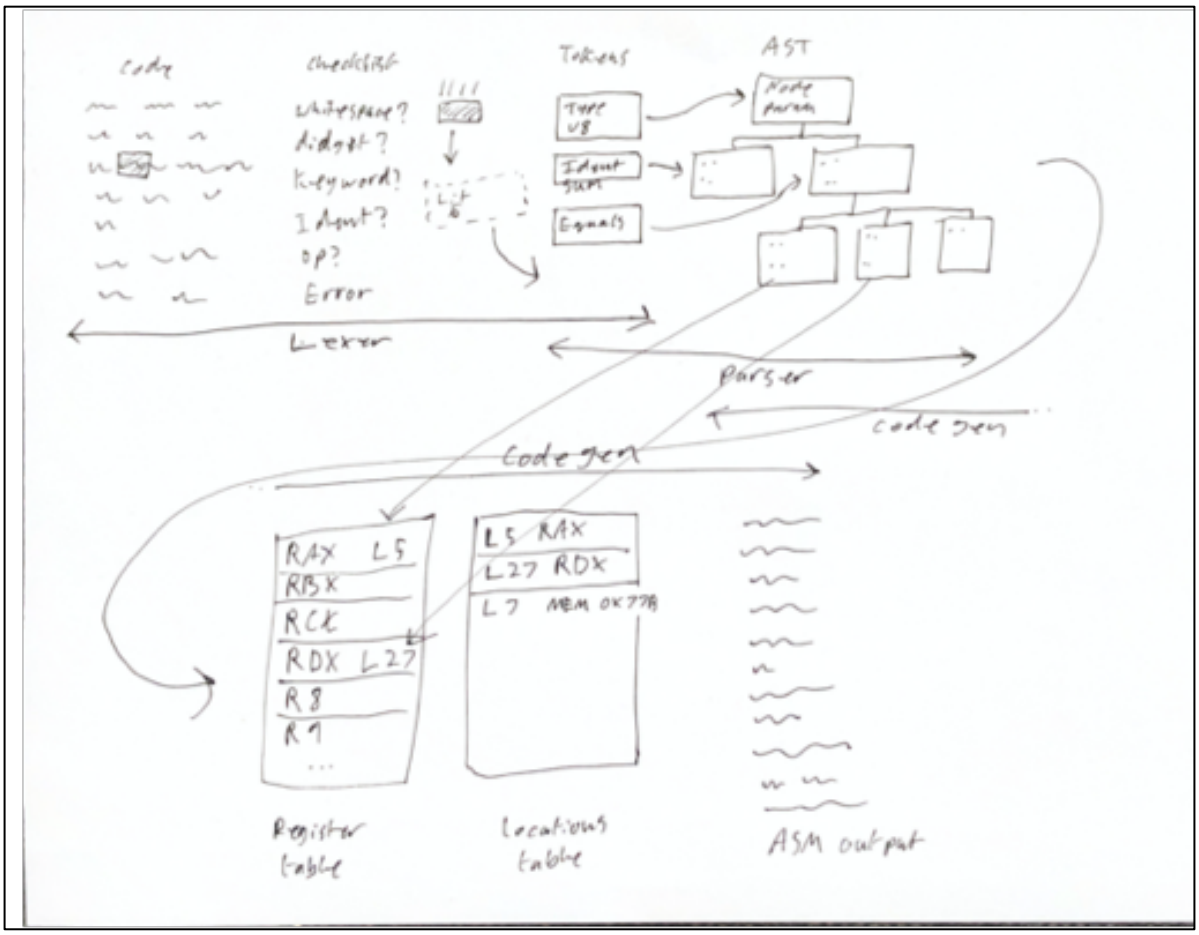

Figure 5 Initial sketch of visualisation flow showing all three phases

During the first sprint the EBNF grammar for the compiler was written, defining the proposed programming language for the compiler and compilation tool. This was further detailed in the Architecture and Design chapter.

Breaking ground on development, a basic project skeleton was created and pushed to the new GitHub repository. The skeleton consisted of both README and LICENCE files along with a C++ main file being built using CMake, the chosen build system of choice for the project. Creating the skeleton project in this sprint allows for the second sprint to be focused purely on its development activities.

6.2 Sprint 2 - Lexer

The aim of the second sprint was to have a functioning lexer for the compiler. To meet the overall project objectives, the compiler must be able to:

- Accept an input source code file
- Output a sequence of tokens
- Process keywords
- Process operator characters
- Process integer numbers
- Skip whitespace

Optionally, as stretch goals, the lexer could be able to:

- Process joint operator characters (e.g. == being a different token from =)
- Process decimal numbers
- Process numbers written in base 16
- Skip single line comments
- Skip multiline comments
- Provide readable error messages on invalid source code

The sprint proceeded with no significant issues and all mandatory objectives were completed. All optional objectives, with the exception of processing base 16 numbers, were completed in addition.

A simple regression test framework was written in the form a bash script that can run a serious of test input source code files and check for a successful run of the compiler. In addition to the test framework, a test file was created to test the lexer and was ran regularly throughout the sprint.

This sprint also contained an investigation spike task to find a suitable graphics library to be used for the visualisation, which will be required from Sprint 5 (note: this was later changed to Sprint 4, as described below). The outcome of this spike was that

Magnum appeared to be the best graphics library for this project, offering both high level abstractions to many common tasks while not specialising significantly in one area of graphics over another[1].

Aside from development activities, the process for gaining ethics approval to perform user research begun.

6.3 Sprint 3 - Parser

The aim of Sprint 3 was to have a functional parser that adheres to the grammar defined in the first sprint. The objectives for the parser are as follows:

- Accept a sequence of tokens
- Output an Abstract Syntax Tree
- Handle parsing of all statements defined in the grammar
- Handle parsing of all expression defined in the grammar[2]
- For expressions to have the correct precedence order in the outputted AST

Optional objectives are as follows:

- Unary operators (e.g. - or + before an integer)
- Provide readable error messages on invalid source code

No significant issues occurred during the third sprint and all objectives, mandatory and optional, we completed. Extra time was available before the close of the sprint so a task from later in the backlog was brought forward and again completed. This additional task was to have the compiler running in a secondary thread in preparation

---

[1] Other potential options focused too heavily on either 3D rendering or interactive user interfaces not capable of graphical visualisations.

[2] Unless they are defined explicitly as an optional objective, in which case they are optional instead.

for the visualisation graphics being implemented, and the appropriate mutexes in place to support thread safe data transfer.

The existing tests for the lexer were updated to handle the parser, in addition to a new test being added to the suite specifically to test the parser. The test suite was ran regularly throughout the sprint.

6.4 Sprint 4 - Event System & Graphics Skelton

The fourth sprint was initially slated to be the completion compiler's code generation phase; however, this was swapped the following sprint's work to change up the type of tasks being developed on as a motivational tactic.

Sprint 4 focused on, as a continuation of the work completed as an additional task in Sprint 3, the creation of an event system to pass data between the compiler and the visualisation front-end. Additionally, there was focus on creating a barebones graphics skeleton that can support the front-end development in future sprints.

The objectives for Sprint 4 are as follows:

- Allow the compiler to post data about its current state
- Store posted data for later retrieval
- Allow the front-end to poll for new or queued events
- Add a graphics library to the build system

Optional objectives for Sprint 4 are as follows:

- Be able to render text and shapes on the screen

This sprint saw the swift development the event system, completing the first three objectives.

Prior to Sprint 4 taking place, the graphics library originally intended to be used was disregarded and replaced with a non-standard choice: Love2d. Integrating Love2d

took a significant amount of time due to the intention to not modify the source code of the library, only applying modifications through the build system to disable certain subsystems of the engine. After the decision was made to forego the nicety of not modifying dependency code, an action that would require ongoing maintenance for a long-lived project (unlike this), the integration with the build system and solution code was quicker than expected. This saw the completion of the remaining mandatory and optional objectives.

6.5 Sprint 5 - Code Generation

The aim of Sprint 5 was to have the compiler generate, from the AST, a valid x86-64 Intel-style assembly file for the NASM assembler. The objectives for the sprint are as follows:

- Accept a completed Abstract Syntax Tree
- Output an assembly file of the specified format
- Track register usage throughout the program
- Process all statements and expressions

The optional objectives for the sprint are as follows:

- Provide readable error messages on invalid source code

Not all of this sprint's objectives were met in full. Two language features were not implemented during the sprint's allotted time: logical AND (&&) and OR (||) operators, and procedure return values. And as would be expected in this scenario, the single optional stretch goal was not completed.

Both of the language features, while initially core in the design for the language, are not significant enough to stop a program from being created if omitted from the language. It is for this reason these features were deprioritised to be completed later if time permits, after all other aim and objectives for the solution have been met. The alternative would have been to roll these tasks into the following sprint, however this

is would result in potential delays to higher priority features. In order to not leave partial functionality spread throughout the solution, the already implemented `&&` and `||` operators were stripped from the Lexer, in addition to the `return` keyword. If time permits in the future to implement code generation for the two language features, the Lexer's code can be recovered easily from the Git history.

The existing tests for the lexer and parser were updated to handle the code generation phase, in addition to a new test being added to the suite specifically to test this phase. The test suite was ran regularly throughout the sprint.

Aside from the two incomplete features, this was a successful sprint as few issues were encountered during development. This is in addition to reaching the important milestone of being able to compile source code to assembly.

6.6 Sprint 6 - Visualise Lexer

The aim of the sixth sprint was to visualise the lexer's data transformations on the screen. The objectives for the sprint are as follows:

- Display the input source code
- Highlight the range of text in the source code that the lexer is currently observing
- Display a checklist of the lexer's character categorisation flow
- Display a list of tokens as the lexer generates them

The sprint's optional objectives are as follows:

- Animate smoothly between states within the visualisation
- Visualise character peeking

This sprint proceeded with no significant issues. All mandatory and optional objectives were met.

Figure 6 Screenshot of the solution's lexing phase

6.7 Sprint 7 - Visualise Parser

The aim of Sprint 7 was to visualise the parser's data transformations on the screen. The objectives for the sprint are as follows:

- Display token list from previous stage
- Highlight the token that the parser is currently observing
- Display the AST nodes as they are being generated
- Display parameters in an AST node

The optional objectives for the sprint are as follows:

- Animate smoothly between states within the visualisation
- Visualise tokens morphing into AST nodes

No significant issues occurred during the sprint and all mandatory objectives were met. The *smooth animation* optional objective was met to a degree that is satisfactory and as far as can reasonably be expected, but was not completed to the same high standard as in Sprint 6. The optional *tokens morphing into nodes* objective was not started.

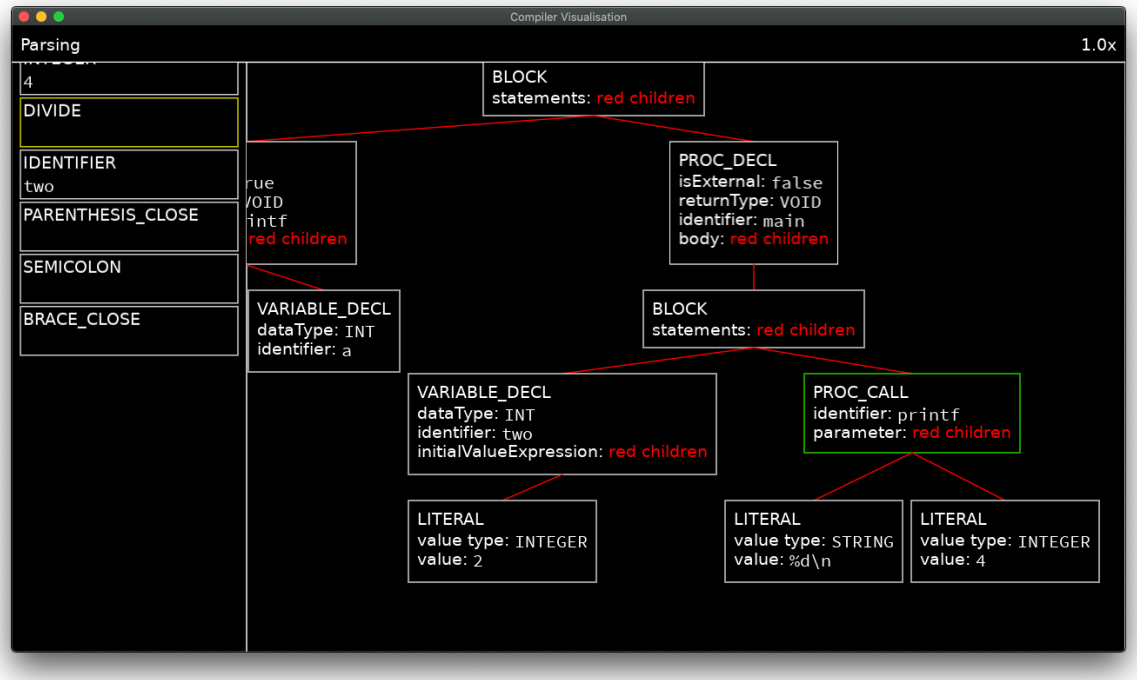

Figure 7 Screenshot of the solution's parsing phase

6.8 Sprint 8 - Visualise Code Generation

The aim of the eighth sprint was to visualise the data transformations in code generation phase on the screen. The objectives for this sprint are as follows:

- Display the AST from the previous stage
- Visualise traversing the AST
- Display a table of the register's contents
- Display a table of the intermediate values' locations

- Display the output assembly file
- Visualise move instructions

The sprint's optional objectives are as follows:

- Animate smoothly between states within the visualisation
- Highlight observed parameter in observed node

This sprint proceeded with no significant issues and all mandatory objectives were met. None of the optional objectives were met due to time constraints.

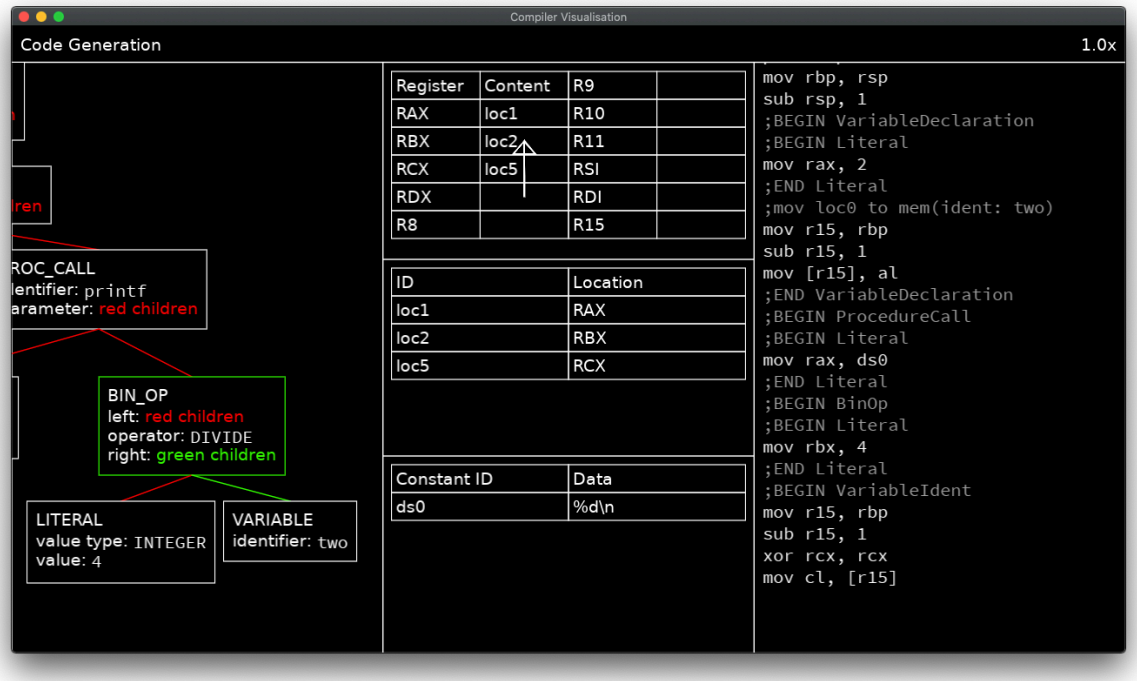

Figure 8 Screenshot of the solution's code generation phase

7 End-Project Report

7.1 Summary

Overall, this project has been a success, meeting the aim of creating a visualisation tool to aid the understanding of how compilers function.

7.2 Project Changes Review

The largest change to the project, deviating from the initial plan, was the lack of formal user research. While ethics approval to conduct user research was sought early in Sprint 2, several delays occurred to this due to bureaucratic technicalities. By the time the ethics approval process was out of the way, the global health outbreak forced the suspension of face to face meetings and the granted ethics approval forebode user research to be carried out virtually over the internet. This temporarily halted the ability to perform user research at a time when it was required for the design of upcoming sprints, namely the visual design for the front-end. While the module's generic ethics approval was subsequently updated to allow for virtual meetings to take place, this occurred at a time which was after the front-end's development, and thus too late[1].

In order to increase motivation for the project, the decision was made to swap Sprint 4 and Sprint 5 around so that some visual elements would be developed sooner, opposed to having three contiguous sprints working on back-end only features. This was of no detriment to delivery but successfully increased motivation.

---

[1] While user research was no longer an option, the ability to perform user testing would have been an alternative validation method that could have been performed if not for the new time constraints imposed on development by the global health outbreak.

As previously mentioned in Sprint 4, the decision was made to replace Magnum with Love2d as the chosen graphics library for the solution. This was primarily around the required learning curve to use Magnum over the simple abstractions of Love2d. Overall this change did not negatively impact the time available for development as Magnum's issues were spotted prior to the start of the relevant sprint.

Two language features initially planned were descoped due to time pressures as detailed in Sprint 5. While this does remove these common language features from the implemented programming language, they are not integral to showcasing how a complier works and complex programs can still be written with this limitation.

7.3 Project Objectives Review

7.3.1 *Develop a custom toy compiler to back the visualisation*

The compiler was developed over four sprints, with the first being planning and architecture design for the sprints to come. During these sprints a functional compiler was created, meeting this objective to allow the user to participate in the visualisation process by providing a source code file for the solution. With the additional benefit of receiving a valid assembly file of their software which they witnessed being created. With negligible exceptions, all the specified mandatory language features were implemented, resulting in the objective being met.

7.3.2 *Develop a programmatic visualisation of a compilation flow*

The visualisation front-end, developed over three sprints, displays the compilation steps clearly, spatially laying out elements and using animation to provide an intuitive reference point for users. The solution's visual aesthetic is in line with the initial design created and maps each of the compiler's fundamental steps. While concessions in

visual polish had to be made in the latter two sprints due to time pressures, the core information from the compiler has still been surfaced to the user. As such, this objective has been met.

7.3.3 *Conduct user research to validate visualisation concepts*

The objective to perform user research was not met due to complications directly incurred from the Coronavirus outbreak. Despite this objective not being met, a complete visualisation flow has been designed and implemented, only it has not sought external user advice. Positive anecdotal user feedback was received colloquially, however this was under no formal user testing arrangement. Even without user research being undertaken, a complete visualisation design and flow was produced that meets the other objective's requirements.

8 Project Post-Mortem

8.1 Project Management Evaluation

Using an agile methodology to manage the project allowed for flexibility during the solution's development, in that detailed architecture was only required for the immediately following sprints. This was important due to the author's lack of knowledge in many of the technologies used and systems being designed, resulting in less confident designs being produced at the start of the project.

While the development generally progressed smoothly, there were moments where sprint scopes and schedules had to be altered. For example, Sprint 4 and Sprint 5 being swapped, and the descoping of procedure returns and logical operators. This kind of flexibility would not have been possible with waterfall, or other such, methods.

8.2 Technologies Evaluation

The technologies used in the solution were the right choice, given the author's background and the size of this project. There is an argument to be made that using a higher-level technology stack would have been more beneficial for the speed of development and visualisation, however considering the author's background lies in low level technologies there is time saved by not necessitating new workflows to be learnt.

Initially choosing Magnum as the graphics library was a mistake due to the learning curve associated with a new framework of this scale. This was noticed before the relevant sprints were started and was rectified with an investigation spike task to replace it. While Love2d was not perfect, since it is not intended for this use case, it happened to provide just the right level of abstraction.

8.3 Developer Performance Evaluation

Averaging the duration of the project, the effort invested was in line with the 400 hour guideline and suitable for the project's scale, adjusting throughout as needed. The initial six weeks of the project saw a planned decline in the expected work effort due to the author's personal commitments. This decrease was planned to be made up for in the second half of the project, when more time would be made available to the author.

The second half of the project saw major disruption due to the global health outbreak, and as such minor planned features had to be descoped, as stated in the Development chapter. In all, aside from the resulting solution, the author has gained invaluable knowledge and experience through the undertaking of this project and is proud of the work achieved.

8.4 Future Work

The solution developed for this project is very much in the proof of concept phase and there is large room for extension and redefinition. For example, if more time were to be allocated to this, the addition of more language features to the compiler would be a good fit. This includes reimplementing procedure returns and logical operators, adding procedure overloading, and adding further semantic analysis to provide better user facing errors. On the visualisation front, user niceties like the addition of a timeline scrubber[1] and more intelligent forward and back skipping shortcuts would be prime features to implement first. Also adding more granular animations and contextual information would aid in the user's grasp of the compilation flow.

[1] Allowing the user to move forward and back in the visualisation, similar to the scrubber in a video playing application.

9 Conclusion

The goal of this project was to create a visualisation tool that can be used to provide insight to people about how a compiler works. This has been achieved since all the solution's minimum requirements have been implemented and the project's aim has been met. While one project objective was not met due to external factors, future work could focus on conducting usability testing and modifying the developed solution based on the feedback.

This project has acted as a foundational basis to deepen my understated of compilers, and helped me learn about lower-level computing fundamentals such as operating system processes and assembly programming.

10 Reference List

Adobe Fonts. 2019. Adobe-Fonts/Source-Code-Pro. [online] Available at: https://github.com/adobe-fonts/source-code-pro [Accessed 10 May 2020].

Arif, S. 2015. VisUAL - A highly visual ARM emulator. [online] Available at: https://salmanarif.bitbucket.io/visual/. [Accessed 25 May 2020].

Atlassian. n.d. Agile? | Atlassian. [online] Available at: https://www.atlassian.com/agile. [Accessed 14 May 2020].

Atlassian. n.d. What is version control | Atlassian Git Tutorial. [online] Available at: https://www.atlassian.com/git/tutorials/what-is-version-control. [Accessed 14 May 2020].

CMake. n.d. CMake. [online] Available at: http://cmake.org. [Accessed 25 May 2020].

Docker. 2020. Docker. [online] Available at: https://www.docker.com/. [Accessed 24 May 2020].

Tool Interface Standard. 1995. Executable and Linking Format (ELF) Specification Version 1.2. Available at: https://refspecs.linuxbase.org/elf/elf.pdf

FreeType. 2018. FreeType Licenses. [online] Available at: https://www.freetype.org/license.html [Accessed 10 May 2020].

GNU Savannah Git Hosting. 2006. FTL.TXT\Docs – FreeType/Freetype2.git - The FreeType 2 Library. [online] Available at: https://git.savannah.gnu.org/cgit/freetype/freetype2.git/tree/docs/FTL.TXT [Accessed 10 May 2020].

Gource - a software version control visualization tool. 2020. Gource - a software version control visualization tool. [online] Available at: https://gource.io. [Accessed 26 May 2020].

GOV.UK. 2019. Agile tools and techniques - Service Manual - GOV.UK. [online] Available at: https://www.gov.uk/service-manual/agile-delivery/agile-tools-techniques. [Accessed 24 May 2020].

Janitor, J., Jakab, F., Kniewald, K. 2010. Visual Learning Tools for Teaching/Learning Computer Networks: Cisco Networking Academy and Packet Tracer. In 2010 Sixth International Conference on Networking and Services (pp. 351-355).

Love2d. 2020. Love2d/Love. [online] Available at: https://github.com/love2d/love [Accessed 10 May 2020].

NASM. 2018. 2.14.02 HTML Documentation. Available at: https://www.nasm.us/xdoc/2.14.02/html/nasmdoc0.html. [Accessed 25 May 2020].

Opensource.org. n.d. SIL Open Font License (OFL-1.1) | Open Source Initiative. [online] Available at: https://opensource.org/licenses/OFL-1.1 [Accessed 10 May 2020].

Opensource.org. n.d. The Zlib/Libpng License (Zlib) | Open Source Initiative. [online] Available at: https://opensource.org/licenses/Zlib [Accessed 10 May 2020].

Python Tutor. n.d. Python Tutor - Visualize Python, Java, C, C++, JavaScript, TypeScript, and Ruby code execution. [online] Available at: http://pythontutor.com/ [Accessed 25 May 2020].

Sangal, S., Kataria, S., Tyagi, T., Gupta, N., et el, 2018. PAVT: a tool to visualize and teach parsing algorithms. Education and Information Technologies, 23(6), pp.2737-2764.

Simple Directmedia Layer. 2020. Simple Directmedia Layer - License. [online] Available at: https://www.libsdl.org/license.php [Accessed 10 May 2020].

White, E., Ruby, J., Deddens, L. 1999. Software visualization of LR parsing and synthesized attribute evaluation. Software: Practice and Experience, 29(1), p.1-16.

11 Appendices

11.1 User Guide

Minimum System Requirements

- macOS 10.14+
- OpenGL
- Git
- Docker for macOS, or Ubuntu and NASM

Instillation

1. Pull down repo: https://github.com/CallumTodd7/compiler-visualization
2. Run `build.sh` to build the application

Running the Application

In the root project directory run: `./bin/cv -i <source_code> -o <output_filepath>`

The `--no-ui` flag can be added to run the compiler without the visualisation.

Running the application will output an x86-64, NASM and Ubuntu compatible, assembly file which can be assembled into an ELF binary. If the visualisation is enabled, a widow will be spawned visualisation the compilation. Press the space bar to start the visualisation.

In order to run the outputted assembly code on a Ubuntu machine:

- Copy the asm file to the Ubuntu machine
- Assemble with: `nasm -g -felf64 output.asm`
- Link with: `gcc -g -no-pie -o output output.o`
- GCC will also include the C standard library
- Run with: `./output`

In order to run the outputted assembly using Docker:

- Copy the asm file to the ./ubuntu_data folder, renaming it to `output.asm` if needed
- Run the `ubuntu.sh` script in the root of the project
- This will open a Ubuntu shell
- Run `./build.sh` to assemble and run the assembly file
- To exit the Ubuntu shell, run `exit`

Window Keybindings

- q/Escape: Quit the application
- Space bar: Start the visualisation
- Space bar: pause/unpause the visualisation
- +: Speed up visualisation
- -: Slow down the visualisation
- 0: Reset the speed of the visualisation
- Right arrow: Skip forward 1 step in the visualisation
- Down arrow: Skip to the next section of the visualisation

Source Programming Language

The application accepts a custom C-style programming language. Below is an example source file that explains the various language features.

```
// This is a comment
```

```
/*
This is a multi-line comment.
Multi-line comments can also be nested as such:
/* Nested comment */
*/


/*
This is an external procedure. The C calling convention is used, so any
C procedures in linked libraries can be used.
*/
extern void printf(void format, int a, int b, int c)


/*
This is a procedure that accepts two parameters.
All procedures should start with `void` because procedure return values
are not supported.
*/
void printSum(int firstParam, int secondParam) {
  // Variable assignment and arithmetic operators (`+`, `-`, `/`, and
`*` are supported)
  int sum = firstParam + secondParam;


  // This is a procedure call.
  // String literals are supported, but strings as a data type are not.
  printf("%d + %d is %d!", sum);
}


/*
The program entrypoint uses the following procedure signature.
*/
void main() {
  int counterA = 5;
  int counterB = 5;


  printf("Printing %d summed numbers using two counters (%d and %d):",
counterA * counterB, counterA, counterB);


  // While statements execute their block while the conditional
expression does not evaluate to zero.
  // Supported relational operators are: `>`, `>=`, `<`, `<=`, `==`,
and `!=`.
```

```
  while (counterA > 0) {
    while (counterB > 0) {
      // If statements execute their block if the conditional
expression does not evaluate to zero.
      // Note: `else` blocks are also supported in an if statement.
      if (counterA != 3) {
        printSum(counterA, counterB);
      }


      // Varaibles can be assigned after they are declared
      counterB = counterB - 1;
    }
    counterA = counterA - 1;
  }


  int counter = 0;
  while (1) {
    counter = counter + 1;


    if (counter >= 10) {
      // `break` will exit out of the current while loop.
      break;
    } else if (counter == 5) { // If statements can occur directly
after the `else` keyword.
      // `continue` will skip to the next iteration of the while loop.
      continue;
    }


    print("The number is %d. %d is less than 10. %d is not 5.",
counter, counter, counter);
  }
}
```

Understanding the visualisation

The visualisation's current section is displayed in the top left of the window. The visualisation's current playback speed is displayed in the top right.

There are three sections to the visualisation: Lexer, Parser, and Code Generation.

Lexer

When lexing: the inputted source code is displayed on the left, with the currently observed characters highlighted; a checklist of the decision tree for the highlighted characters is displayed in the middle; and the outputted stream of tokens is displayed on the right.

Two colours are used when highlighting characters: magenta being a peeked character, and yellow being the current selection.

Parser

When parsing: the inputted stream of tokens is displayed on the left; and the Abstract Syntax Tree intermediate representation is displayed on the right. Each node in the tree displays the node type as the first line of text, followed by any parameters the node has. A parameter may either have a text value or a child node, in which case it will be colour coded. Child nodes are colour coded to the parent node's parameter.

Code Generation

When in the code generation phase: the inputted Abstract Syntax Tree is displayed on the left; the register table is displayed in the top middle; the locations tables is displayed in the middle; the constants table is displayed in the bottom middle; and the outputted assembly instructions are displayed on the right. The register table is responsible for showing register availability, with the left-hand column listing the registers and the right-hand column listing the register's contents. The locations table is responsible for showing all variables and intermediate values currently in use. The constants table is responsible for showing a list of all literal values (string literals or integer literals). Comments in the outputted assembly are coloured grey.

Notices

11.2 Project Management – Trello Progression

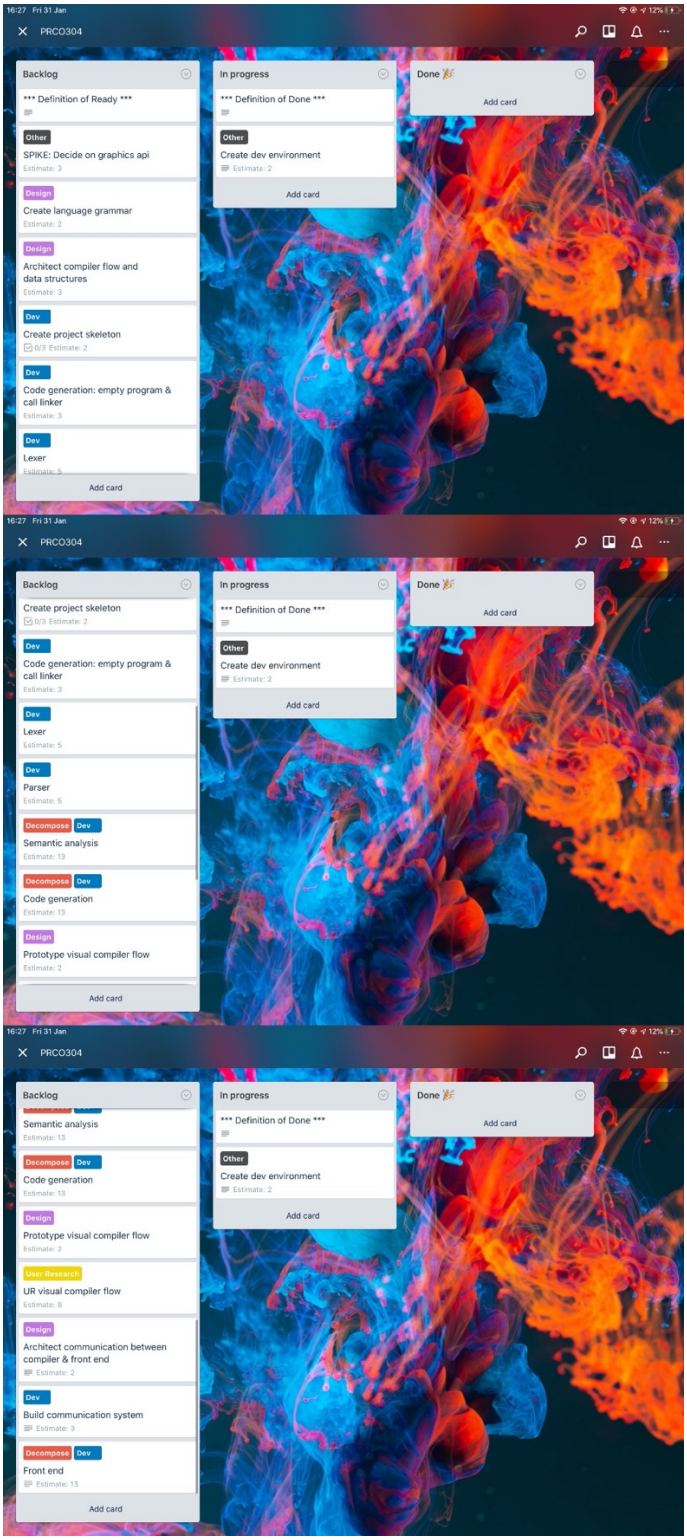

Figure 9 Trello screenshot – Sprint 1

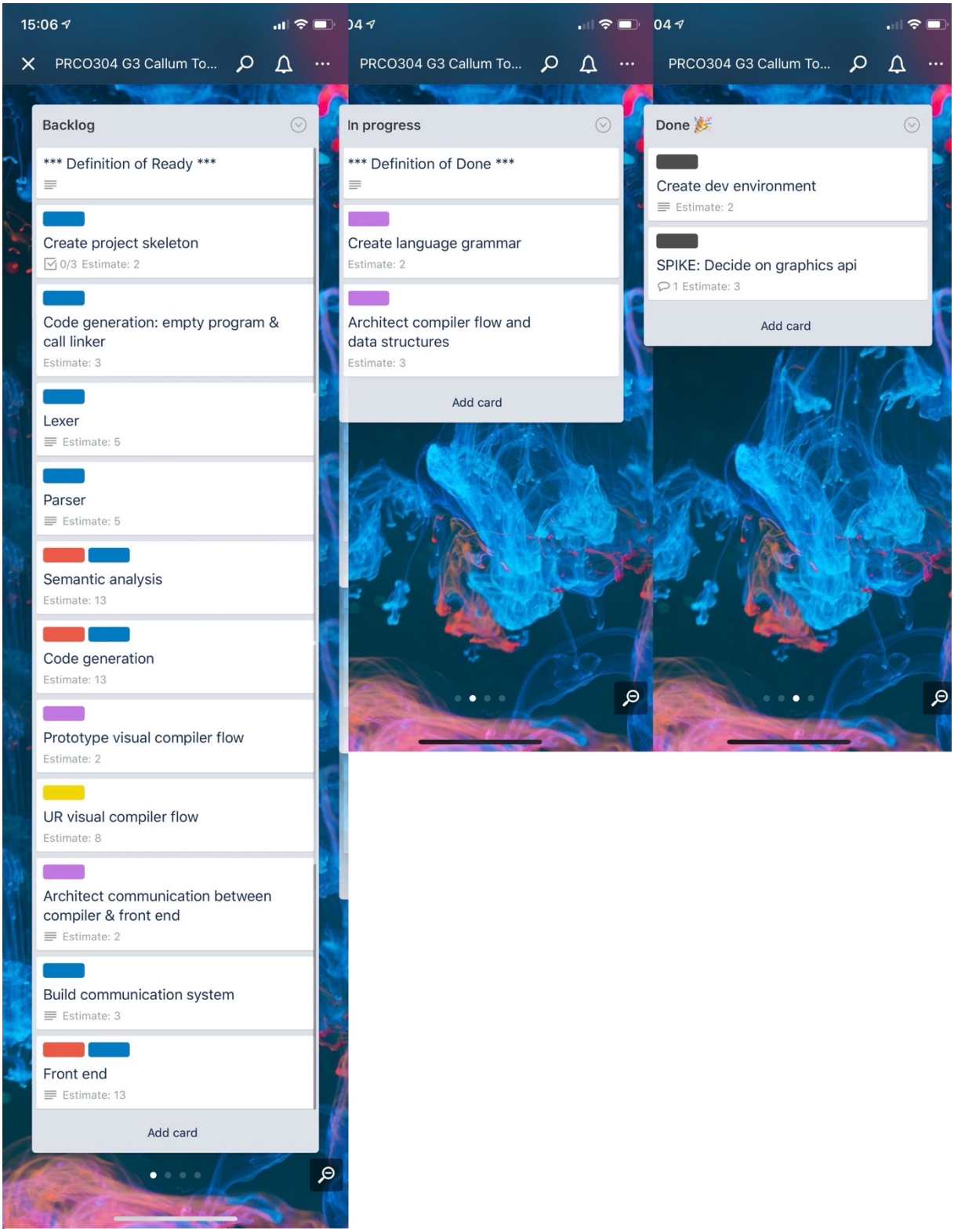

Figure 10 Trello screenshot – Sprint 2

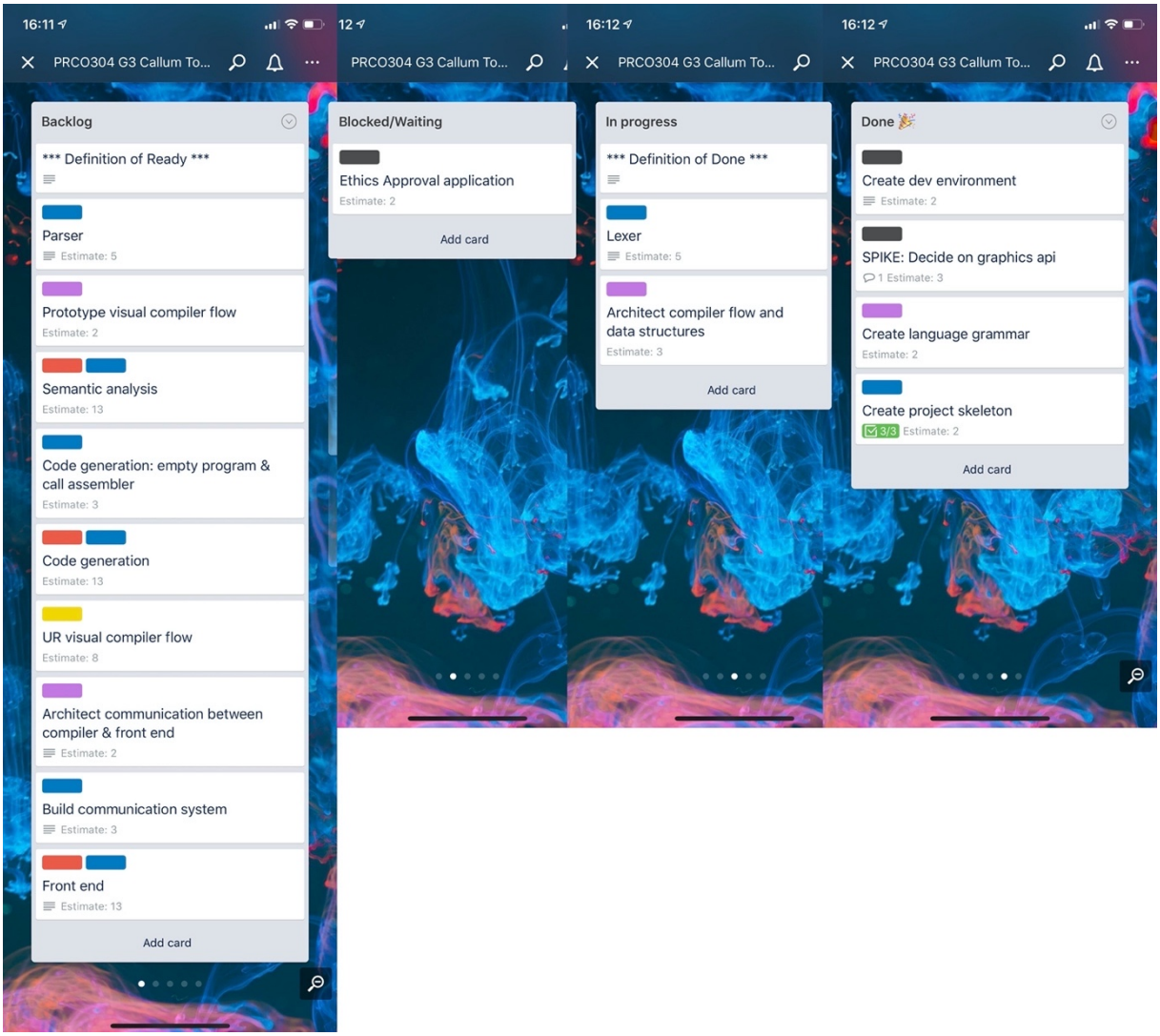

Figure 11 Trello screenshot – Sprint 3

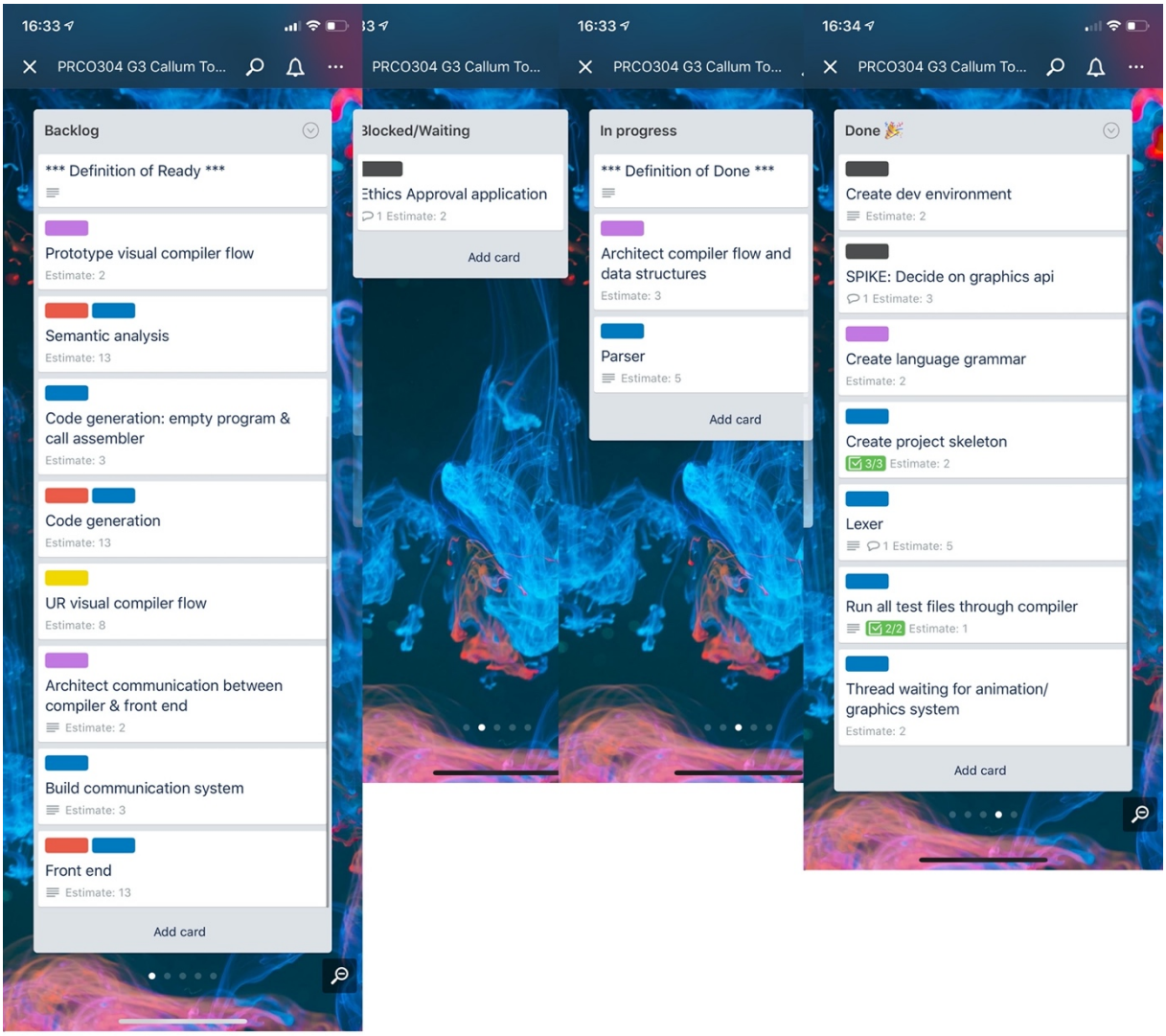

Figure 12 Trello screenshot – Sprint 4

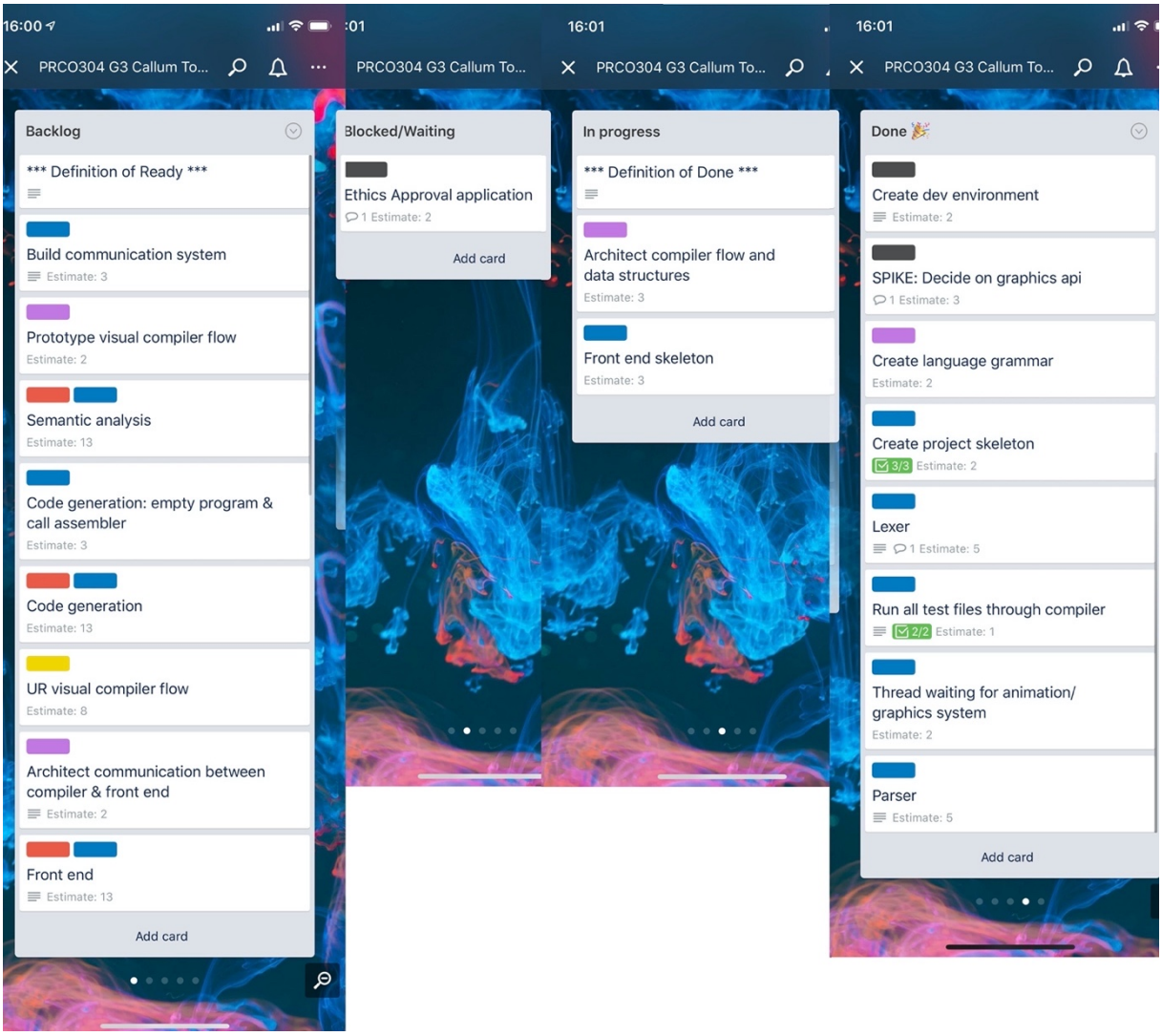

Figure 13 Trello screenshot – Sprint 5

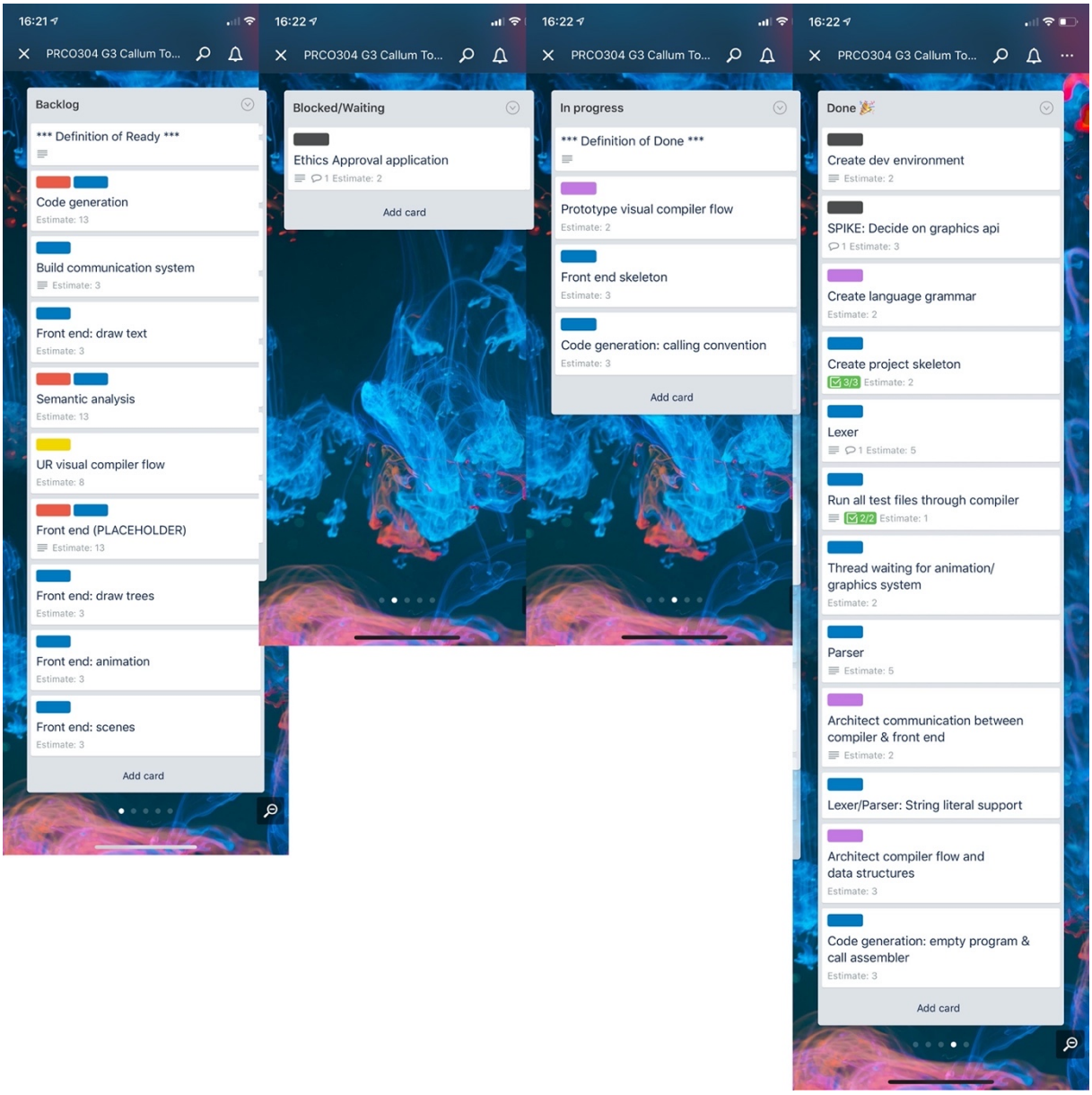

Figure 14 Trello screenshot – Sprint 6

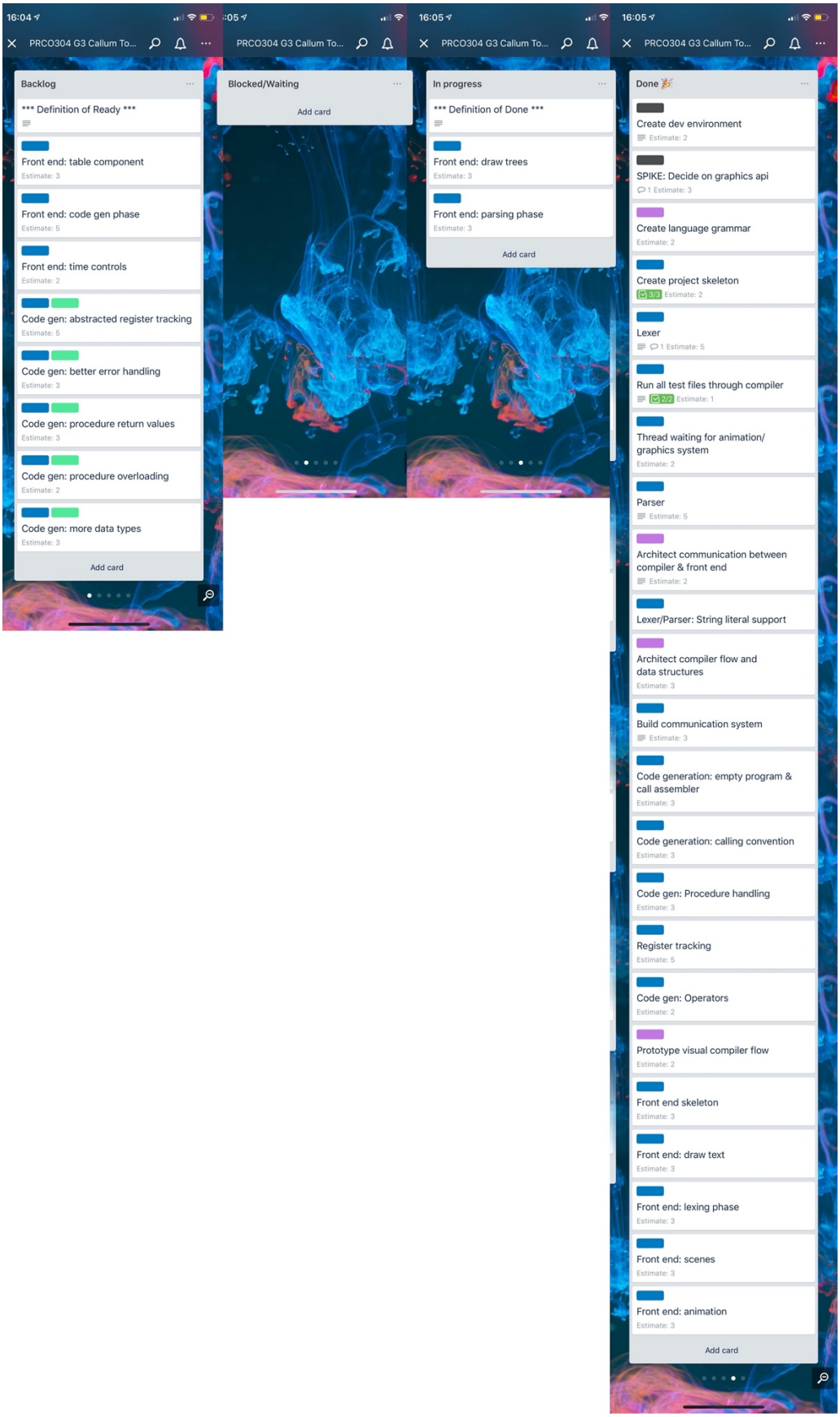

Figure 15 Trello screenshot – Sprint 7

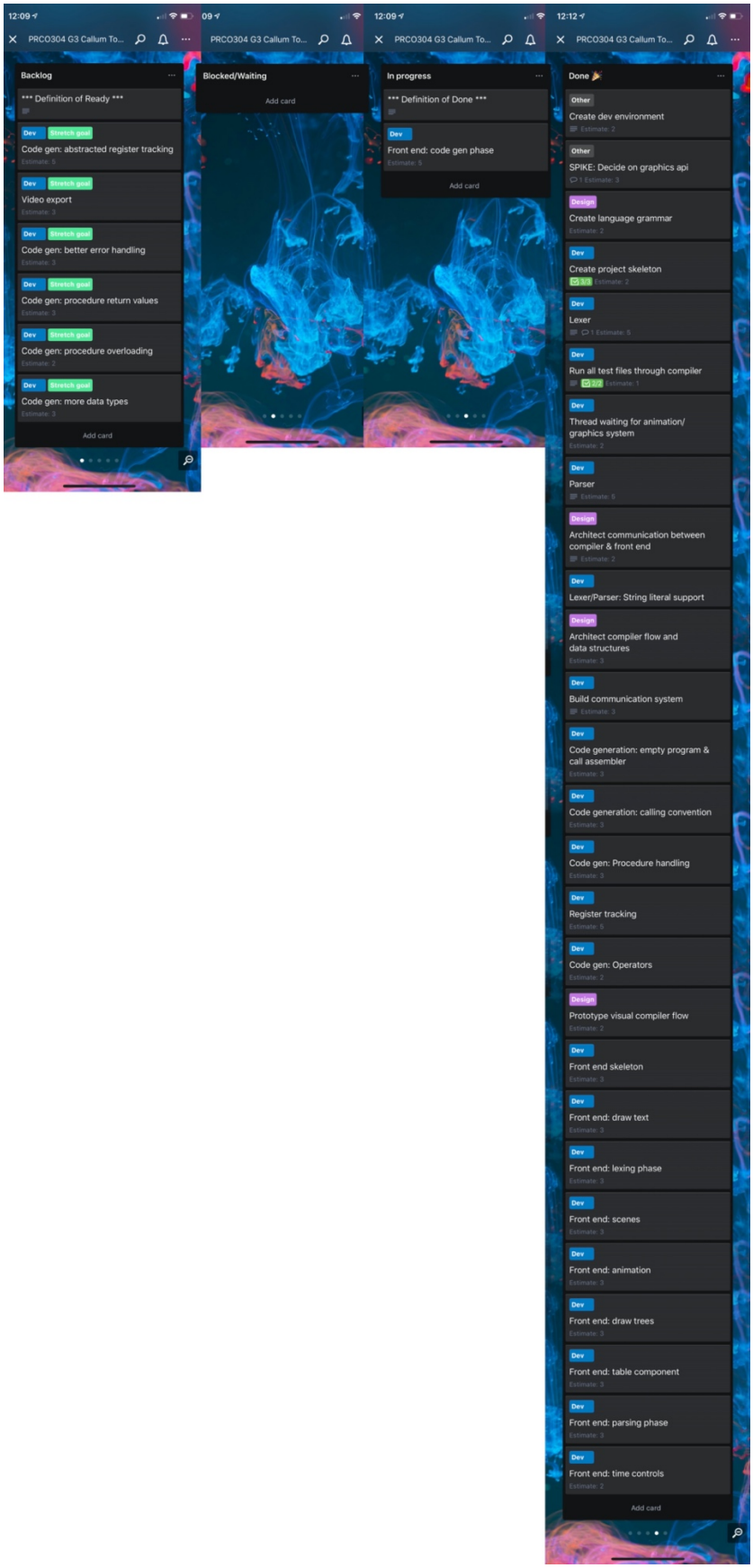

Figure 16 Trello screenshot – Sprint 8

11.3 Language EBNF Grammar

```
digit
= "0" | "1" | "2" | "3" | "4"
| "5" | "6" | "7" | "8" | "9"
;


letter
= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
| "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
| "U" | "V" | "W" | "X" | "Y" | "Z"
| "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
| "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
| "u" | "v" | "w" | "x" | "y" | "z"
;


int_const
= {digit}
;


identifier
= letter, {letter | digit | "_"}
;


variable
= identifier
;


type_specifier
= void
| int
;


factor
= int_const
| variable
| "(", expression, ")"
;


unary_factor
= ["+" | "-"], factor
```

```
;


term
= unary_factor, [{ ("*" | "/"), unary_factor }]
;


expression
= term, [{ ("+" | "-"), term }]
;


iteration_statement
= "while", "(", expression, ")", block
| "for", "(", [expression], ";", [expression], ";", [expression], ")",
block
;


selection_statement
= "if", "(", expression, ")", block, ["else", (block |
selection_statement)]
;


jump_statement
= "continue"
| "break"
| "return", [expression]
;


statement
= iteration_statement
| selection_statement
| jump_statement
| variable_declaration_statement
| variable_assignment_statement
| block
;


block
= "{", [{statement, ";"}], "}"
;
```

```
variable_declarator
= type_specifier, identifier
;


variable_declaration_statement
= variable_declarator, ["=", expression], ";"
;


variable_assignment_statement
= identifier, "=", expression, ";"
;


function_declaration
= type_specifier, identifier, "(", [variable_declarator, [{",",
variable_declarator}]], ")", block
;


program
= {function_declaration}
;
```