Rios

Reactive Input/Output State-Machines

Simon Vinberg Andersen,
Sebastian Hjorth Hyberts,
Mathias Knøsgaard Kristensen,
Simon Svendsgaard Nielsen,
Felix Cho Petersen,
Shahab Shajarat

Spring 2019

AALBORG UNIVERSITY

STUDENT REPORT

Rios

Reactive Input/Output State-Machines

Group Name:

sw404f19

Supervisor:

Ingo van Duijn

Group Members:

Simon Vinberg Andersen
Sebastian Hjorth Hyberts
Mathias Knøsgaard Kristensen
Simon Svendsgaard Nielsen
Felix Cho Petersen
Shahab Shajarat

Project Page Count:

78

Total Page Count:

101

Denne rapport udforsker muligheder og problemstillinger forbundet med at udvikle et programmeringssprog og tilhørende oversætter, til brug ved input/output-baserede problemer. Omdrejningspunktet for projektet er en undren vedrørende nuværende redskaber, der anvendes i forbindelse med mikrocontroller miljøet. Dette gøres via en analyse, der belyser både mikrocontrollere som platform, men også styrker og svagheder i de associerede programmeringssprog. Endvidere benyttes dette som basis for skabelsen af et sprog til håndtering af en specifik problemtype. Denne process beskrives i omfattende stil, og designvalg taget i løbet af processen forklares — både for valg truffet under designet af programmeringssproget, men også for implementeringen af oversætteren. Yderligere, på baggrund af relevante test gennemført, bliver der konkluderet på hvorvidt det skabte programmeringssprog reelt håndterer problemer på tilfredsstillende vis, i henhold til de krav der blev opstillet i løbet af projektet. Der konkluderes at Rios er en færdig compiler, i den forstand at det kan kompilere programmer udelukkende via syntaks defineret i Rios. Til slut vurderes der forskellige mulige tilføjelser til sprogets tilgængelige produktioner, og hvordan disse ville være gavnlige.

Preface

This report is written by sw404f19 during the fourth semester on the bachelor's degree in software, during the spring of 2019. The purpose of the project is to focus on creating a new programming language, a compiler for the language, and understanding the underlying concepts that are required to do this. A special thanks is given to Ingo Van Duijn, our councillor on this project, and Hans Hüttel, our syntax and semantics lecturer.

Aalborg Universitet, May 27, 2019.

Simon Vinberg Andersen
svan17@student.aau.dk

Sebastian Hjorth Hyberts
shyber17@student.aau.dk

Mathias Knøsgaard Kristensen
matkri15@student.aau.dk

Simon Svendsgaard Nielsen
ssni17@student.aau.dk

Felix Cho Petersen
fcpe17@student.aau.dk

Shahab Shajarat
sshaja17@student.aau.dk

Reading Guide

If a citation is before a period mark (.), the citation is for the line it is on exclusively.

The purpose of this section is to highlight citation in practice. ***As such, the sentence in bold, is the sentence that is cited*** [1].

If a citation comes after a period mark (.), the citation is for the entire paragraph.

The purpose of this section is to highlight citation in practice. As such an example has been created with the intention of showing this in practice. [1]

The Context Free Grammar in this report is using Extended Backus-Naur Form where the quotation marks (") are omitted and replaced with blue bold font when showing syntax text.

Contents

Preface IV

1. Introduction 5
   1.1. Introduction to Microcontrollers . . . 5
   1.2. Personal Experiences with Microcontrollers . . . 5
   1.3. Introduction to Arduino . . . 6
   1.4. Initiating Problem Statement . . . 6
   1.5. Summary of Chapter 1 . . . 7

2. Analysis 8
   2.1. I/O-based Problems . . . 8
   2.2. How to Solve I/O-based Problems . . . 8
      2.2.1. I/O Automaton . . . 9
      2.2.2. Arduino Language and I/O-Problems . . . 11
   2.3. Language Evaluation Criteria . . . 11
   2.4. Evaluation of C++ and the Arduino Language . . . 14
      2.4.1. Evaluation of C++ . . . 14
      2.4.2. Evaluation of the Arduino Language . . . 15
   2.5. Juniper . . . 16
   2.6. Problem Statement . . . 19
      2.6.1. Limiting the Problem Area . . . 20
   2.7. Summary of Chapter 2 . . . 20

3. Rios Language Specification 21
   3.1. Language Requirements . . . 21
      3.1.1. Functional Requirements . . . 21
      3.1.2. Non-Functional Requirements . . . 22
      3.1.3. Design Choices . . . 22
   3.2. Syntax and Semantics . . . 25
      3.2.1. Abstract Syntax and Helping Functions . . . 25
      3.2.2. State . . . 27
      3.2.3. Reaction . . . 33
      3.2.4. Reaction Collision . . . 34
      3.2.5. Variables . . . 38
      3.2.6. Statements . . . 40
      3.2.7. Expressions . . . 41

3.2.8. Transition . . . 46
3.2.9. Types . . . 48
3.2.10. Static Scoping . . . 49
3.3. Summary of Chapter 3 . . . 50

4. The Compiler 52
4.1. Understanding the Compiler . . . 52
4.1.1. Understanding the Rios Compiler . . . 53
4.2. The Rios Lexer . . . 55
4.3. Lexer Generating Tools . . . 56
4.4. Picking a Parsing Approach . . . 56
4.4.1. Ambiguity . . . 57
4.4.2. LR(1) Parser . . . 58
4.4.3. LL(1) Parser . . . 58
4.4.4. Comparing Parsing Techniques . . . 59
4.5. Parser Generating Tools . . . 59
4.5.1. ANTLR . . . 59
4.5.2. COCO/R . . . 60
4.5.3. Manual . . . 60
4.6. The Rios Transformer . . . 60
4.7. Making a Symbol Table . . . 61
4.7.1. Utilising Visitor Patterns . . . 62
4.8. Reaction Sorting . . . 62
4.9. Code Generator . . . 63
4.9.1. Reactions . . . 63
4.9.2. States . . . 64
4.9.3. Transitions . . . 65
4.10. Summary of Chapter 4 . . . 67

5. Evaluation 68
5.1. Practical Test . . . 68
5.2. Language Evaluation Criteria . . . 71
5.3. Discussion . . . 72
5.4. Conclusion . . . 73
5.5. Future Work . . . 73
5.5.1. Extensive Testing Suites . . . 74
5.5.2. User-functions . . . 75
5.5.3. Control Structures . . . 75
5.5.4. Type Coercion . . . 75

Bibliography 76

Appendices 79

A. Smoke/Gas Detector Code 80

B. Context Free Grammar 82

C. Structural operational semantics 85
C.1. Meta-sets . . . 86
C.2. Helping Functions . . . 86
C.3. Transition Systems . . . 87
C.4. Declaration . . . 88
C.5. Global . . . 91
C.6. Transition . . . 92
C.7. Local . . . 93
C.8. Expression . . . 94

D. Type rules 97

E. Code Examples 101
E.1. DigitalReadSerial Example - Arduino Language . . . 101
E.2. DigitalReadSerial Example - Rios . . . 101

1. Introduction

The initial basis for this project was based in a general curiosity regarding microcontrollers. Through experience gained via prior projects, a few limitations were discovered, when applying the Arduino Language to certain problem domains. This project intends to explore whether these limitations are representative for a significant part of microcontrollers. To begin exploring this, a certain knowledge base has to be created. As such, this chapter will introduce various concepts related to microcontrollers to help readers understand the fundamental idea of how they function.

1.1. Introduction to Microcontrollers

A microcontroller unit is a small computer housed on a single integrated circuit (IC), that works based off of a user-defined program. Microcontrollers function by utilising simple sequences wherein a machine code instruction is given, decoded, and then executed to perform a given hardware operation [1, p.15]. At the core, a microcontroller is a miniature processor, and as such, uses vary greatly, making them a very versatile tool. They are very diverse, and an abundance of different platforms exists with a multitude of ways to program them [2]. However, microcontrollers are typically used as input/output (I/O) devices to connect and control, external devices and components. I/O applications often involve an external device sending data to the microcontroller via pins. The data is processed and, if it fulfils certain user-defined criteria, activates procedures within the microcontroller. These procedures cause certain reactions; either by activating an external device or by triggering internal functions. [3]

1.2. Personal Experiences with Microcontrollers

Half of the project members had personal experiences with microcontrollers, specifically Arduino, prior to this project. The commonality in all cases was that the initial contact occurred during high school education, where one of the purposes was to teach both programming and electronics. One of the observations was that the Arduino Language often required frequent repetition of code between projects, especially code concerning things like pin setups or functionality when signals from a pin were activated. In larger projects, this became especially apparent. Utilising the object-oriented programming of C++ could also be very restricted when writing larger programs, because of the limited

available program memory on the Arduinos used in the courses. This sometimes resulted in projects implementing state-machines to restructure the programs, often causing the same state-machine implementations to be repeated numerous times. Prior knowledge with regards to Arduino, and the platform being widely used, was also the cause for the focus of this project on said brand from the start.

1.3. Introduction to Arduino

Arduino is an open-source microcontroller platform utilising easy-to-operate hardware and software [4]. At its core, the Arduino platform usually incorporates an IC from the ATMega line of microcontrollers made by Atmel Corporation [5]. Most Arduino boards give access to a number of pins for signals, a USB port for communicating with and uploading programs to said board, as well as a port for supplying power [5]. A significant part of Arduino programs are programmed in a fashion, where a set of functions are kept in a state of waiting, until they are triggered by a specified value measurement by an external device. This assumption is built upon the contents of the Arduino Project Hub [6] at the time of this reports conception. As such I/O-based applications are a core facet of many Arduino Language applications [7].

Additionally, Arduino has its own language, called the Arduino Language. It is one of the most commonly used language for programming on Arduino microcontrollers. This is due to built-in libraries containing Arduino specific functions. However, Arduino Language is but one of multiple usable languages for Arduino that each fill specialised needs that the general Arduino Language does not cater to, or is meant as replacements for the Arduino Language.

The Arduino Language itself is a dialect of the C++ language, meaning it uses large parts of the C++ language while also employing others of original design. The differences in that Arduino Language often comes down to simplifications of C++ meant to make the Arduino Language easier to approach than C++ (this is explained further in section 2.4.2).

1.4. Initiating Problem Statement

Arduino is a platform meant to allow even the most inexperienced programmers, to incorporate microcontrollers and associated hardware with a user-defined piece of software, in a manageable manner [7]. As was mentioned in section 1.3, a significant amount of available Arduino projects were centred around input-output based situations. Given that the Arduino Language is built upon C and C++, both of which are general purpose languages, it can be assumed that the Arduino Language is as well. As such, it could be argued that the Arduino Language caters to a generality, which in turn might lack more specialised functionality for addressing I/O-based problems.

In order to further explore the dynamic between the general Arduino Language, and the handling of a specialised subset of problems, the following initiating problem was formulated to guide the problem analysis:

> *"How well is a general computational language, such as the Arduino Language, suited for a domain centred around I/O-based problems?"*

As such focus should be on understanding what I/O-based problems entail, as well as the potential strengths of a specialised language. Additionally, the initiating problem also reflects that Arduino is a general-purpose language that has many uses, and as such warrants an evaluation method for determining a given language specialation for specifically I/O-based problems.

1.5. Summary of Chapter 1

This chapter introduced microcontrollers as a concept. This was done by explaining the theory behind their composition, as well as give a brief look into their uses. The chapter also introduced the Arduino brand of microcontrollers, and delved into the personal experiences that was used as a basis for picking Arduino as the target platform for Rios. Finally, this chapter presented a question, to serve as the basis for an analysis into the I/O-based problems and their interactions with the microcontroller platform.

By reading this chapter readers should acquire a firm understanding of what a microcontroller is, as well as why the language being developed within this report is to be modelled for the purpose of handling problems on the Arduino platform. Additionally, readers should be aware of where this focus comes from, in addition to why the analysis in chapter 2 is conducted.

2. Analysis

In order to explore and answer the specified initiating problem statement, it is necessary to expand upon the concepts presented in the first chapter. As such, further concepts, related to input/output-based problems, are introduced and discussed. This includes, analysing what I/O-based problems encompass, in addition to what the typical procedures are for these types of problems. Given the focus of the domain, projects specifically written to run on Arduino microcontrollers are chosen as the archetype, and as such, will be the ones to be analysed. This should help garner an understanding as to which requirements the analysed languages should be able to fulfil, in order to be labelled as an I/O specialised language, in addition to how well it fulfils them. For this purpose, we will introduce several evaluation criteria for which to analyse a language. Using these should lead to an answer for the initiating problem statement.

2.1. I/O-based Problems

In this report, an I/O-based problem refers to a problem in which a microcontroller needs to communicate with the outside world. This process should be done by having I/O processors handle the transference of data between peripheral devices and components, such as a microcontroller and an LED for instance [8].

2.2. How to Solve I/O-based Problems

As previously mentioned in section 1.1, I/O-based problems are a significant part of the problems solved by using microcontrollers. Therefore, a number of Arduino projects based on I/O were analysed, one of which was a project revolving around a smoke detector. The source code to this can be found in appendix A. The problem is simple: measure a value and change the behaviour when the measured value exceeds a given threshold. If the threshold is exceeded, a red LED should light up and a tone should play from a buzzer, otherwise a green LED should be lit and the buzzer should be silent.

2.2.1. I/O Automaton

According to Lynch[1] and Tuttle[2] from Massachusetts Institute of Technology, we can model this problem using the input/output automaton model. The model can be described using five components:

- an action signature $sig(A)$,
- a set $states(A)$ of $states$,
- a nonempty set $start(A) \subseteq states(A)$ of $startstates$,
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$,
- an equivalence relation ***part*** (A) partitioning to set ***local*** (A) into at most a countable number of equivalent classes.[9]

The last of the components, the equivalence relation, has been omitted in order to simplify the solution, since it is only relevant to fairness computation which is an important factor in multithreading. However, Arduino boards generally cannot commit to multithreading due to the built-in specs of most Arduino boards not being able to handle it. The action signature consists of all actions in the system, these actions are classified as one of three types, ***input actions, output actions,*** and ***internal actions***. Input actions are the actions in which the environment gives information to the system, whereas output actions send out information to the environment. Lastly, internal actions are simply actions which the outside world can not perceive [9].

Using these components we can now model the previously mentioned smoke detector problem. We define the set of action signature as follows:

Input actions: PUSH

Output actions: RED, GREEN, BUZZER

Internal actions: none

The states of the smoke detector consist of a ***polling state*** and an ***alarm state***. Each of these have access to the same variables ***red***, ***green*** and ***buzzer***, - all of which can take on the values 0 or 1 relating to an on or off mode - a set ***threshold*** and a ***reader*** which both can take on a value which can be represented by an integer. The ***polling state*** is the start state where the variables ***red***, ***green*** and ***buzzer*** are set to 0, 1, 0 respectively, the ***threshold*** is whatever value the user specified, and the ***reader*** is 0. Similarly, the ***alarm*** state initially sets the variables ***red***, ***green*** and ***buzzer*** to 1, 0, 1 respectively. The transition relation of the smoke detector is described by stating a precondition and an effect for every action π, in case the precondition is simply $true$ it will be omitted.

PUSH

E□ect: $reader \leftarrow input\_value$

[1]Professor in Software Science and engineering at MIT,
[2]PH.D Student as of the papers release

RED

Precondition: $red = 1$

E□ect: Turns on red LED

GREEN

Precondition: $green = 1$

E□ect: Turns on green LED

BUZZER

Precondition: $buzzer = 1$

E□ect: Turns on buzzer

It should be noted that the input actions π reaches a high number of actions, in this case, as it can simply be thought of as any integer being a separate input action, due to it simply being a value read from the outside world. For this reason we have limited the input actions to be represented by two distinct actions, namely, $aboveTreshold = \{x \mid x \in \mathbb{Z}, x > threshold\}$ and $belowTreshold = \{y \mid y \in \mathbb{Z}, y \leq threshold\}$. Where all input values in $aboveTreshold$ would jump to the polling state, and all input values in $belowTreshold$ would jump to the alarm state. This dynamic can also be modelled as shown below.

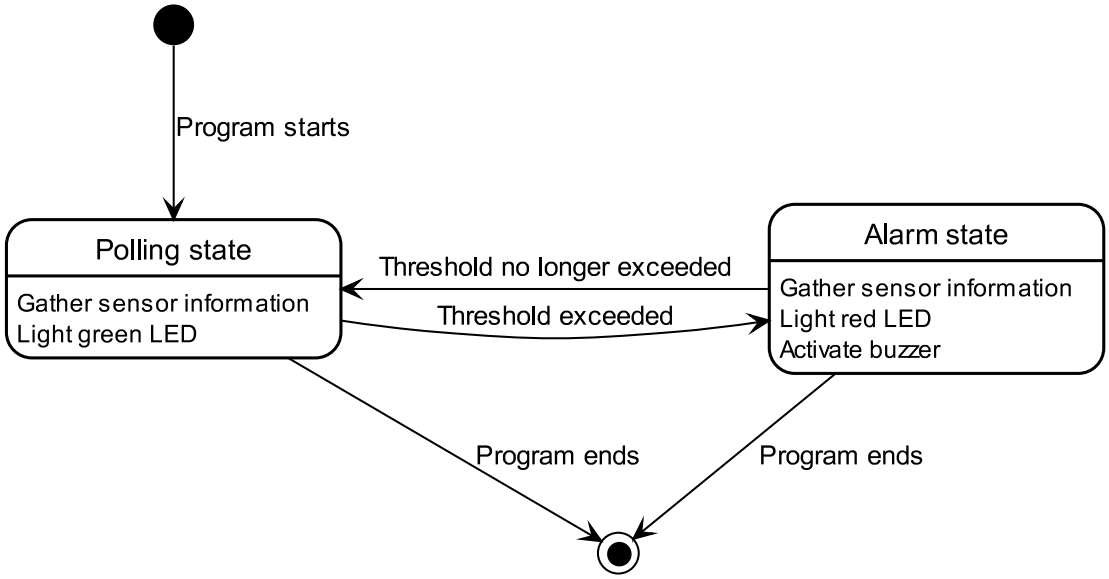

Figure 2.1.: Two-state diagram of the smoke detector

We would start in the polling state, then repeatedly read the value from the outside world and incorporate either $aboveTreshold$ or $belowTreshold$, whichever is relevant.

2.2.2. Arduino Language and I/O-Problems

Using the applied solutions in section 2.2 as a baseline for what a language centred around I/O-based problems should be able to simulate, reveals a need for the addition of certain concepts.

While a general computational language such as Arduino is able to implement the highlighted solutions, it requires heavy manual work from the user. This is because the generality of the language ensures the user must specify as much as they can, while the compiler predicts and assumes as little as possible, in order to cater to the most possible use cases. This means that using a general computational language such as Arduino to solve strictly I/O-based problems likely includes a redundant amount of generality, which is not befitting these domain-specific problems.

2.3. Language Evaluation Criteria

In order to evaluate the Arduino Language, a special consideration should first be taken with respect to I/O-based problems, namely evaluate how well the language takes advantage of relevant aspects of the problem domain. In the case of the presented solutions in section 2.2, states, transitions, and actions all play an integral part in solving these types of problems. However, in order to properly evaluate how *well* the language incorporates these concepts, it is paramount that we first define what makes a language *good* in the first place. As such, certain language evaluation criteria are utilised in order to carefully examine the fundamental parts of good programming languages.

For this report, the method of evaluation is based in the evaluation rule-set presented within the the book 'Concepts of Programming Languages' by Robert W. Sebesta . The language evaluation criteria within Sebesta's book is split into four overarching categories: readability, writability, reliability, and cost. An analysis will be conducted on the Arduino Language to both explain the underlying concepts of the language evaluation criteria, as well as evaluating strengths and weaknesses of the Arduino Language with respect to solving I/O-based problems.

Readability evaluates the ease with which a given language can be read and understood. Additionally, readability must only evaluate the language within the context of the intended domain model [10]. As such, in the problem domain of I/O-based problems, given a source code, the easier to identify and understand concepts such as states, actions and transitions the better the readability. Similarly, given a I/O-based problem, how easily the programmer would be able implement the appropriate states, actions and transitions would determine the Writability of the programming language [10]. Reliability is measured by how well a program fulfils required specifications for all possible situations. Reliability relies on both readability and writability to succeed. A general language that does not support the required approaches to a problem within the problem domain, will likely utilise a sub-optimal approach that is less likely to be universally applicable [10].

Lastly, the cost is measured by the cost effectiveness of utilising a language - measured both in money and time. It is measured by 7 criteria: [10]

- The cost required to train programmers in the use of the language.
- The time required to write a program in the language
- The cost of compiling a program in the language
- The cost of executing a program in the language.
- How well the language implementation system is designed.
- How reliable the system is, with regards to running properly at critical junctions.
- How easy it is for a programmer, that is not the author, to maintain software utilising the language.

In addition to the aforementioned four primary categories, there are nine sub-categories that are utilised to evaluate specific sub-problems within a given category. The various sub-categories can be seen in Table 2.1.

	Readability	Writability	Reliability
Simplicity	*	*	*
Orthogonality	*	*	*
Data types	*	*	*
Syntax design	*	*	*
Support for abstraction		*	*
Expressivity		*	*
Type checking			*
Exception handling			*
Restricted aliasing			*

Table 2.1.: A table that depicts which sub-categories three of the major categories used to evaluate a language rely on. Like in the book, cost has been omitted, as it is evaluated on a different set of rules. Table taken from 'Concepts of Programming Languages' page 31 [10].

Simplicity evaluates the ease with which a programmer can pick up the language and learn it. It evaluates various problems related to having multiple solutions for the same problem, the less options the simpler the language [10]. This is an important criterion to consider when dealing with domain-restricted problems. As certain concepts are already accepted in certain domains, in the case of I/O-based problems these have proven to be concepts such as states, actions and transitions (see section 2.2). Therefore having the user manually incorporate a state, for instance, in one of multiple possible ways adds unnecessary complexity, compared to simply having the language support states as a concept, with a single way of incorporating it into the program solution.

Orthogonality is the concept of combining primitive data types and functions to make more complex data types. Higher degrees of orthogonality make a language more simple to use, but too much orthogonality can also make it complex. As an example of orthogonality, the C language has some simple data types in ints, floats and chars. Combined with the pointer data function, these can be turned into arrays if the data is stored in consecutive memory. [10] Given I/O-based problems, certain data types could be useful when considered primitive. Given that an I/O problem is required to communicate with the outside world, having pins and serial connection, the primary way of communication for the microcontroller, as primitive data types might prove useful in contrast to the user manually constructing them.

Data type evaluation measures a languages ability to define data types and structures adequately. A type system is used to associate types with values to increase readability. If a language lacks necessary types, like a boolean type, it would constitute using alternatives such as 0 and 1, which could muddle the intention and needlessly complicate the code. [10] Given the domain of I/O-based problems, a data type such as states would seem ideal for solving these kinds of problems.

Syntax design evaluates the simplicity and readability of the written part of a given language. It evaluates different aspects, such as keywords, as well as the form and meaning of statements. It should evaluate how well compound statements are designed, whether keywords are locked so as to not be usable as variable names, as well as whether the semantic meaning is gaugeable from the keyword. [10]

Abstraction as a concept is the idea of representing an entity using only its most significant attributes. For example, one could define birds to be animals with two legs, two wings, a tail, and feathers. Abstraction presents itself primarily through process abstraction and data abstraction. Process abstraction encompasses subprograms or functions, while data abstraction encompasses objects and their resulting data types. [10]

Support for abstraction would thus mean that a program to some degree supports subprograms, objects, or both. As expressed within section 2.2, a potential solution for solving I/O problems is to utilise state machines. This solution would entail having various states that a program can jump between. States would consist of a set of actions, that would only be relevant, and need representation when utilised. As such there could be a very limited type of data abstraction present.

Expressivity is the evaluation of convenience of expressing computations through code. Many languages have shorthand versions of doing certain actions, to avoid cumbersome and bloated code. A common one is a post-increment operator. Instead of writing the actual addition piece, many languages support the shorthand of simply writing *variable++* to increment a variable by 1. [10] In an I/O setting it might prove that certain actions are common, and therefore adding a short hand for such expression could increase the specialisation of the language, to even further cater to specifically I/O-based problems.

Type checking is employed to test for type errors. In contrast to data type evaluation, type checking evaluates whether types are used for their intended purpose, and not

whether the right types are present in the language. As such, different approaches to type checking exist. Examples of this could be by checking all parameters at compile time, or some combination of formal and actual parameter checking. [10]

Exception handling is the ability for a program to intercept run-time errors, and handle them in such a way that the program may be able to continue. Some languages offer extensive exception handling (Ada, C++, Java, C#), while others offer little to none (C, Rust, Go). [10] In many languages, exception handling consists of setting up try/catch block. Imperative languages like C could potentially implement something resembling exception handling, but it generally does not fit within the languages scope. When a problem is occurred, it stops the program execution, and tries to find a catch block to solve the error. If a catch block is not found, the exception escalates through the call stack. This does not stop until either a catch block is found, or in the case of no catch blocks, an error message is printed by the program.

Aliasing is the concept of having two or more distinct names in a program that all access the same memory cell. For example, two pointers that both point to the same memory location. [10]

2.4. Evaluation of C++ and the Arduino Language

With respect to the initiating problem in section 1.4 an evaluation of the Arduino Language will be presented. However, as it fundamentally is a dialect of C++ with some specialised libraries, an evaluation of C++ will first be conducted. After this, the evaluation will be expanded to include the parts unique to Arduino Language. The method for evaluating the two languages will use the criteria described in section 2.3.

2.4.1. Evaluation of C++

C++ is an extension of the C language that adds a lot of new things to make programming easier, while also using a wide range of data types to secure clear and concise programming syntax. It supports many data types, ranging from the built-in Boolean to the user-defined enumerations, and as such provide programmers with tools for a significant part of possible use cases. It allows the primitive data types along with pointers to these. Additionally, it introduces objects, allowing for further abstraction and further orthogonality. It splits these data types into three overarching categories: built-in, user-defined, and types derived from fundamental types. Additionally, C++ allows for some semblance of aliasing, through the use of pointers, where users make multiple variables that point to the same memory space. Though it adds many new types and allows many new operations on these types, C++ maintains some of the simple syntax from C. The most significant difference is the presence of classes within the language. Another significant difference is the means of using libraries. The calls to library functions in C++ follow the syntax `libraryName::desiredFunction()`. It should be noted, that with the

very large amount of standard libraries included in C++, the language itself is also considered quite large. This can make it harder to learn, as there could be some features a new programmer would need, but would not know how to find.

C++ inherits some C expressions that increase expressivity, such as pre- and post-increment operators. The large amount of standard libraries also add their own expressions for the functionality they concern, making it easier to write more complicated computations in the language. To facilitate this expressivity, and to allow for various actions, C++ also includes exception handling. This is done through try/catch blocks, though a thrown exception may propagate up through the call stack as long as it is not caught. While an extension of C, making compilation slower, it can also be used with a relatively small feature-set. As such, one can start small in C++ and slowly begin expanding the number of features being used. This allows programmers to start small, but also risks having two programmers learn two different subsets of C++.

2.4.2. Evaluation of the Arduino Language

We can extend the evaluation of the C++ language somewhat by recognising that most of the Arduino Language is contained in libraries that are made available by the IDE. Most of these libraries are concerned with making it easier to work with different aspects of Arduino usages. The major ones concern pins and pin modes, while others concern things such as servo motors, EEPROM memory, Ethernet connections, among others [11]. These libraries fulfil the input/output concerns for Arduino boards specifically. The issue of modelling state machines, however, persists in the language, as none of the Arduino standard libraries concern modelling state machines.

Typically, a programmer set to program a microcontroller will use an accompanying IDE, if available [12]. These IDEs contain many functions to make programming easier, including pin interaction, copying compiled code to the memory of the controller, among other functions. In addition to these functions, the Arduino IDE removes much "boilerplate" code that would be found in normal C++ programs. This is especially noticeable in the removal of the need for function prototypes and library imports. These are automatically inserted during compilation of a program sketch.

First it can be assumed that a C++ programmer uses an appropriate library for their given microcontroller. Thus it can also be assumed that they will use the Arduino IDE for programming the Arduino controller. As such, some of the downsides from pure C++ is mitigated through the use of the tools in the IDE.

	C++	Arduino
Simplicity	Somewhat simple, but size of language limits simplicity	Much boilerplate removed, simplifying programs
Orthogonality	Somewhat orthogonal	Somewhat orthogonal
Data types	Good amount of primitive types and type operations. Objects allow further data types	Good amount of primitive types. Libraries allow connected components to be used as data types
Syntax design	Somewhat simple syntax. Some parts are obscure until a user learns about them (function prototypes, library calls)	Simple syntax, especially when facilitated by standard libraries.
Abstraction	Objects allow a great deal of abstraction.	Objects are not commonly used, but are available. Most abstraction is done through libraries and their types.
Expressivity	Some operations can be done in many ways. Operator overriding allows for further expressivity.	Some operations can be done in many ways. Operator overriding allows further expressivity, though not for library functions.
Type checking	Statically typed and scoped.	Statically typed and scoped.
Exception handling	Simple exception handling with try/catch blocks.	No exception handling.
Aliasing	Pointers allow aliasing. Objects also allow aliasing to an extent.	Pointers allow aliasing.

Table 2.2.: Evaluation summary

2.5. Juniper

When analysing any given problem domain, a look at possible solutions to the problems can help inspire or deter from recreating an already existing solution. During this analysis, a number of arguments have been laid out to identify the lack of specialisation for I/O-based problems in the Arduino language, despite it being a primary use case for Arduino use cases. This same conclusion was already reached by the creators of Juniper, a functional reactive programming language (FRP), who explicitly states in their

documentation that,

> [...] many Arduino programs are reactive: they respond to incoming signals, process those signals, and generate new output signals. Using the existing C++ environment, these programs quickly turn to "spaghetti" code that lacks modularity and is difficult to reason about.[13]

Juniper allows users to employ a range of high level features like parametric polymorphic functions and immutable data structures. Juniper was made to handle timing based events that, according to the creators, the current Arduino Language is not suited to handle. In this section, an example of a simple program written in the Juniper language will be presented.

```
1  module ButtonDebounce
2  open(Prelude, Button, Io)
3
4  let buttonPin = 2
5  let ledPin = 13
6
7  let bState = Button:state()
8  let edgeState = ref Io:low()
9  let ledState = ref Io:high()
10
11 fun button() = (
12         let buttonSig = Io:digIn(buttonPin);
13         let debouncedSig = Io:fallingEdge(Button:debounce(buttonSig,
               ↪ bState), edgeState);
14         let ledSig =
15                 Signal:foldP(
16                         fn (event currentLedState) ->
17                                 Io:toggle(currentLedState)
18                         end,
19                         ledState, debouncedSig);
20                 Io:digOut(ledPin, ledSig)
21 )
22
23 fun setup() = (
24         Io:setPinMode(ledPin, Io:output());
25         Io:setPinMode(buttonPin, Io:input());
26         Io: digWrite(ledPin, !ledState)
27
28 fun main() = (
29         setup();
30         while true do
31                 button()
```

```
32          end
33  )
```

Listing 2.1: A program that outputs toggles an LED when a button is pressed

The code seen in Listing 2.1 describes a program in the Juniper language that continuously polls a button for an input, and toggles an LED on and off. The program was pulled from the example site of Juniper. We will explain the program in further detail below.

```
1   module ButtonDebounce
2   open(Prelude, Button, Io)
3
4   let buttonPin = 2
5   let ledPin = 13
6
7   let bState = Button:state()
8   let edgeState = ref Io:low()
9   let ledState = ref Io:high()
```

Lines 1 to 9 are simply variable declarations.

```
10  fun button() = (
11          let buttonSig = Io:digIn(buttonPin);
12          let debouncedSig = Io:fallingEdge(Button:debounce(buttonSig,
                ↪ bState), edgeState);
```

From line 10 a function is defined. `buttonSig` is set up to read digitally from the previously-defined `buttonPin`. `debouncedSig` is given as a debounced signal from a button, specifically from the falling edge of the signal when the button is pressed.

```
13          let ledSig =
14                  Signal:foldP(
15                          fn (event currentLedState) ->
16                                  Io:toggle(currentLedState)
17                          end,
18                          ledState, debouncedSig);
19                  Io:digOut(ledPin, ledSig)
20  )
```

Line 13 to 20 finish the function definition. The variable `ledSig` is given as the result of a function `foldP`. `foldP` takes as input a lambda function (lines 16 to 18), a variable (`ledState`) and a signal (`debouncedSig`). When the debounced signal is activated, `foldP` will use the lambda function to toggle the LED state. Line 20 outputs the desired LED state to the appropriate pin.

```
21      fun setup() = (
22          Io:setPinMode(ledPin, Io:output());
23          Io:setPinMode(buttonPin, Io:input());
24          Io: digWrite(ledPin, !ledState)
25
26  fun main() = (
27          setup();
28          while true do
29                  button()
30          end
31  )
```

Lines 21 to 31 simply set up the program and run the button function indefinitely.

It should be noted, that as of Juniper 2.2, the `Signal:foldP` function can be replaced with a built-in `Signal:toggle`, providing the same functionality. In the example program, this saves 4 lines of code. [14]

2.6. Problem Statement

After having analysed I/O-based problems and what they entail, we based the evaluation criteria in a theory of I/O automatas which should solve our domain-specific problem set. Certain concepts, like states, actions and transitions were of particular interest, and were assigned special focus in an attempt to determine whether or not a general computational language, such as Arduino, is suited for solving I/O-based problems. Given the lack of specific I/O-based datatypes and apparent specialisation, it was concluded that a general computational language was not suited. It should be noted, that while we conclude that the general computational language approach is not an appropriate approach, it does not mean it is unable to solve the domain-restricted problems, rather that these solutions are unsatisfactory. As such, a problem statement has been formulated in order to solve the issue,

> *Given that I/O automata are a common possible solution for problems solved with Arduino microcontrollers, how can a language be designed which is based in the relevant aspects of the automata, and specialised for Arduino microcontroller boards?*

Given that a part of the identified problems with the analysed languages were a distinct lack of I/O specific concepts and models, the problem statement has been formulated as a means to address that issue. The solution to the problem statement should be a new language that easily, and intuitively implements the necessary and specialised

concepts mentioned in section 2.2 with the explicit purpose of dealing with I/O problems in microcontrollers.

2.6.1. Limiting the Problem Area

Given that the problem statement explicitly requests a new language, it is assumed that it is new language, and not an extension to an already existing language. As such, the design and complexity of the new programming language would expectantly vary greatly depending on the scope with which it is developed with. These concepts will be explored further in chapter 3, and an outline will be given for what the scope of the language entails.

2.7. Summary of Chapter 2

This chapter presented the general problem that had been found in chapter 1 and the solution that had been found. This chapter then delved into why a general language might not be suited to handle specialised problems. This was done by presenting certain problems present within the Arduino Language, and then proposing potential improvements based on these. Arduino Language was subjected to extensive evaluation, to properly analyse what limitations, with respect to I/O-based problems, could be. Because of the fact that Arduino Language consists of C++ with a few specialised libraries, the analysis was conducted first on C++ and then the specialised Arduino libraries. These, in conjunction with an analysis of a current solution, Juniper, served as the basis for locating strengths and weaknesses of the microcontroller platform at present. Finally, all of this was condensed into a single problem, that is meant to help guide the reader throughout the rest of the report.

By reading this chapter, readers should be aware of the limitations found within the C++ and Arduino Language, as well as in some specialised languages like Juniper. Readers should have gained insight into why general-purpose languages might, in some cases, not be designed to deal equally well with every type of problem. Finally, readers should have gained an understanding as to why utilising automata could be beneficial to an input/output-based platform.

3. Rios Language Specification

To fulfil the goal specified within section 2.6, Rios needs some semblance of specialisation that already available solutions do not provide. To achieve this goal, certain constructions and productions must be defined and given semantic meaning, to create a better-suited solution to I/O-based problems, than what was evaluated within section 2.3.

Within this chapter all syntax and related semantics for Rios will be presented. In addition to the meaning behind the language constructs of Rios, the chapter also holds a definition of the expectations the authors behind the language hold. These are based in both personal experiences as programmers, but also in the language analysis conducted within Chapter 2.

3.1. Language Requirements

In section 2.3 a set of evaluation criteria was set forth, which served as an evaluation method to determine how specialised a given language is in regards to I/O systems. Therefore, a number of requirements will be presented, which if implemented successfully, should assist in reaching a high level of specialisation. In the coming section, a set of requirements, separated into functional- and non-functional requirements, will be listed. These requirements were formulated based primarily on the I/O automata presented in section 2.2, in order to ensure the language will support every relevant concept mentioned therein. The requirements will be used throughout the design phase, in an effort to guide the process, as well as being included within the final language evaluation.

3.1.1. Functional Requirements

Functional requirements is part of the system development process. They explain what the intention of the language is and what Rios should be able to do. As such, they revolve around evaluating specific operations like calculations, data manipulation, and technical details. [15] The functional requirements for Rios are as follows:

1. Rios should be able to solve I/O based problems using specialised concepts relevant to model an I/O automaton. This includes but is not limited to:
    a) States
    b) Start State

c) Actions

d) Transitions

2. Rios will be able to compile and be uploaded to Arduino boards.
3. Rios should support operations to relevant data types. This includes:
   a) Arithmetic operations
   b) Logical operations
4. During an compilation error, the compiler should give the exact location of the dysfunctional code.

3.1.2. Non-Functional Requirements

Non-functional requirements, like functional requirements, are established, when creating a system, to help evaluate whether it fulfils its specified goals. Non-functional requirements are used to evaluate the operation as a whole, rather than the specific function evaluation that functional requirements deals with. These evaluation criteria can be, but are not limited to, the time between critical failure occurrences, to the cost of maintaining necessary facilities - to name a scant few. [15] The non-functional requirements formulated for Rios are:

1. It should be possible to identify specialised concepts relevant to model an I/O automata. This includes:
   a) States
   b) Start State
   c) Actions
   d) Transitions
2. The design choices for Rios should not be limited by a specific Arduino microcontroller model.

3.1.3. Design Choices

In order to simulate an I/O automaton, Rios should implement the relevant concepts highlighted in section 2.2. Furthermore, the language needs to be evaluated in accordance with the evaluation criteria presented in section 2.3, meaning the implementation should also be done simply and effectively. Given that states and actions are singular concepts, minimal modifications were necessary in order to respect the evaluation criteria. However, given that a singular action can be one of three action types, namely an internal action, input action or output action, Rios modifies the concept of actions into

a singular new concept of reactions. To recap the action types, an internal action should not be observable from the outside world, it should be executed entirely internally. This is in contrast to input and output actions, which can be observed. An input action is the act of having the system gather information from the outside world, for instance having a button pressed down, or measuring a given value from the environment. Similarly, an output action is the act of having the system pass information to the outside world, for instance by printing a message or shining a light [9]. By analysing the structure of each action type (see figure 3.1) a simplification can be made.

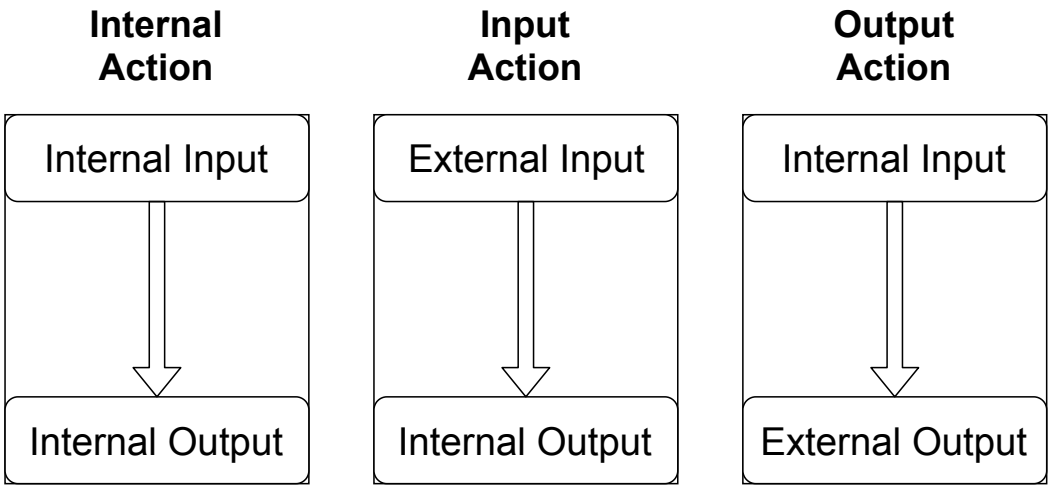

Figure 3.1.: Visual representation of the three different action types in an IO automata

We notice that the actions types can be connected in the sense that:

- An input action begins with an external input and ends with an internal output.
- An internal action begins with an internal input and ends with an internal output
- An output action begins with an internal input and ends with an external output.

By letting the internal output from the input action function as the internal input of the internal action, and similarly the internal output of the internal action as the internal input of the output action, the concept of the reaction can be realised.

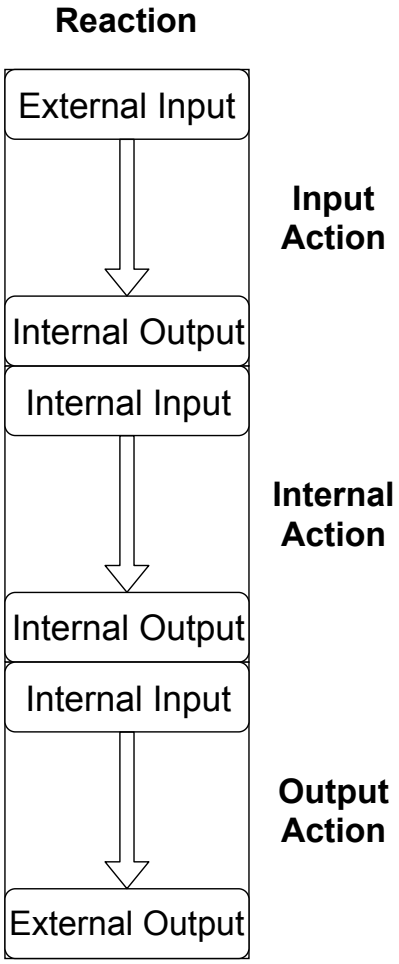

Figure 3.2.: Visual representation of a reaction in Rios

However, this means that every internal action will be dependant on an input action which might not be desirable in some cases. For example, imagine the scenario where a lamp needs to turn on by the press of a button and then turn off again after 5 seconds. In this instance, there is no input action to turn off the light, only an internal action which determines whether or not it is time to turn off the light, and then an output action which turns said light off if prompted. For this reason, parts of a reaction in Rios can be omitted, meaning a reaction can exist independent from an external input, or exist without any external output to show the outside world. This further allows Reactions to dynamically interact, as they can now extend each other. This means given a singular external input, a singular external output can be given, but after being processed by multiple internal actions in different reactions. It should then be noted that this versatility means it is possible to solve other problems than strictly IO based in Rios, as the given program may not be dependant on environmental actions in order to execute actions.

3.2. Syntax and Semantics

The following section will explain the Rios language specification, focusing on single concepts at a time. The relevant parts of the Context Free Grammar for the concept will be shown, along with an informal semantic explanation, followed by structural operational semantics for the concept in question.

3.2.1. Abstract Syntax and Helping Functions

Before the language specification can be understood, an abstract syntax and some helping functions will be presented. The helping functions following the abstract syntax will be used extensively in the coming sections, and generally represent some smaller part of the program or program state.

The concrete syntax, abstract syntax, and operational semantics can also be found in their entireties in Appendix B and Appendix C

Abstract syntax categories

n	$\in \mathbf{Num}$	Numerals
x	$\in \mathbf{Name_V}$	Variable Names
s	$\in \mathbf{Name_S}$	State Names
Dp	$\in \mathbf{Dec_P}$	Priority List Declarations
Do	$\in \mathbf{Dec_O}$	OnEnter Declarations
Dv	$\in \mathbf{Dec_V}$	Variable Declarations
Ds	$\in \mathbf{Dec_S}$	State Declarations
Dr	$\in \mathbf{Dec_R}$	Reaction Declarations
Dc	$\in \mathbf{Dec_C}$	Reaction Case Declarations
S	$\in \mathbf{Stmt}$	Statements
e	$\in \mathbf{Expr}$	Expressions
u	$\in \mathbf{Unit}$	Time units
T	$\in \mathbf{Type}$	Types
p	$\in \mathbf{Prog}$	Program

Abstract syntax

$$
\begin{aligned}
e &::= n \mid x \mid \texttt{(}\ e\ \texttt{)} \mid e_1\ \texttt{+}\ e_2 \mid e_1\ \texttt{-}\ e_2 \mid e_1\ \texttt{*}\ e_2 \mid e_1\ \texttt{/}\ e_2 \\
&\quad \mid e_1\ \texttt{==}\ e_2 \mid e_1\ \texttt{!=}\ e_2 \mid e_1\ \texttt{<}\ e_2 \mid e_1\ \texttt{>}\ e_2 \mid e_1\ \texttt{<=}\ e_2 \mid e_1\ \texttt{>=}\ e_2 \\
&\quad \mid \texttt{!}e \mid e_1\ \texttt{||}\ e_2 \mid e_1\ \texttt{\&\&}\ e_2 \\
Dp &::= s\texttt{,}\ Dp \mid s \mid \varepsilon \\
Do &::= \texttt{onenter:}\ S \mid \varepsilon \\
Dv &::= t\ x\ \texttt{=}\ e\ Dv \mid \varepsilon \\
Ds &::= \texttt{state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds\ \texttt{\}}\ Ds \mid \varepsilon \\
Dr &::= \texttt{when}\ c \mid \varepsilon \\
Dc &::= \texttt{|}\ e\ \texttt{:}\ S\ c \mid \varepsilon \\
T &::= \texttt{apin} \mid \texttt{dpin} \mid \texttt{serial} \mid \texttt{int} \mid \texttt{long} \mid \texttt{bool} \mid \texttt{byte} \mid \texttt{float} \\
S &::= S_1\ \texttt{;}\ S_2 \mid x\ \texttt{=}\ e \mid Dv \mid \texttt{transition(}\ s_1\ \texttt{)} \mid e \mid \varepsilon \\
p &::= Ds
\end{aligned}
$$

Meta sets

$$v \in \mathbf{Val} = \mathbb{Q} \cup Strings \cup \{true, false\} \tag{3.1}$$

$$r \in \mathbf{Rea} = (\mathbf{Dec_C} \times \mathbf{Var} \times \mathbf{Pri}) \tag{3.2}$$

$$r^* \in \mathbf{Rea}^* \tag{3.3}$$

$$next_s \in \mathbf{Name_S} \times \{\varepsilon\} \tag{3.4}$$

$$curr_s \in \mathbf{Name_S} \tag{3.5}$$

$$l \in \mathbf{Loc} \tag{3.6}$$

$$\mathbf{Name_{S\mathcal{E}}} = \mathbf{Name_S} \cup \{\varepsilon\} \tag{3.7}$$

The set **Val** denotes the possible values of variables in Rios. This consists of all rational numbers, any string of characters, and the (boolean) values true and false. The set **Rea** denotes the possible reactions as the cartesian product of all possible case declarations and all variables. $next_s$, $curr_s$, and l represent variables used for the semantics. Respectively, they represent the next next state in transitions, the current state of the program, and a location.

Helping functions

$$\mathcal{N} : \mathbf{Num} \to \mathbb{Q} \tag{3.8}$$

$$pri \in \mathbf{Pri} : (\mathbf{Name_S} \cup \{next\}) \rightharpoonup \mathbb{N} \tag{3.9}$$

$$var \in \mathbf{Var} : \mathbf{Name_V} \rightharpoonup \mathbf{Loc} \tag{3.10}$$

$$sto \in \mathbf{Sto} : \mathbf{Loc} \rightharpoonup \mathbf{Val} \tag{3.11}$$

$$sta \in \mathbf{Sta} : \mathbf{Name_S} \rightharpoonup \underbrace{\mathbf{Rea}^*}_{Reactions} \times \underbrace{\mathbf{Stmt}}_{OnEnter} \times \underbrace{\mathbf{Name_{S\mathcal{E}}}}_{Parent} \times \underbrace{\mathbf{Name_{S\mathcal{E}}}}_{DefaultChild} \tag{3.12}$$

$$anc(s) = \{s\} \cup anc(pare_s) \textbf{ where } (pare_s, ...) = sta(s) \tag{3.13}$$

$$rel \in \mathbf{Rel} : (\mathbf{Rea} \times \{reader,\ writer\}) \rightharpoonup \mathbf{Loc}^* \tag{3.14}$$

$$nextsto(l) = l + 1 \tag{3.15}$$

The helping functions provide various information for the semantics as needed. The function $\mathcal{N}$ denotes the conversion from numerals to numbers. The function *pri* is a mapping from a state name to a natural number representing the priority of a state. This is used for checking state priorities. The functions *var* and *sto* together denote a variable name and its corresponding location and value. The function *sta* maps a state name to a tuple of possible reactions, OnEnter statements, as well as the name of the parent state and the name of the default child state of the given state, both of which has the option to be empty. The function *anc* takes a state name as input and returns the set of state names denoting the given states ancestors. This is mostly used to ensure that the correct OnEnter statements are executed. The function *rel* gives a reaction and its relation to a given location or number of locations, so that reactions can later be sorted, as will be described in section 3.2.4 and section 4.8. Finally, the *nextsto* function simply extends storage as needed.

3.2.2. State

$$\begin{aligned}
Decs &::= \{Dec\} \\
Dec &::= VarDec \\
&\mid ReactDec \\
&\mid StateDec \\
&\mid OnEnter \\
&\mid PriorityList \\
StateDec &::= [\texttt{default}]\ \texttt{state}\ StateId\ \texttt{\{}\ Decs\ \texttt{\}} \\
StateId &::= bigLetter\ \{anyFollowing\}
\end{aligned}$$

States are environments that determine which Reactions are active and which Variables are in scope, they also have an optional OnEnter and a mandatory Priority List.

(DEC-STATE)

$$\frac{\begin{array}{r} var, rel, pri, pare_s \vdash \langle Dp,\ pri\rangle \rightarrow_{Dp} \langle pri'\rangle \\ s \vdash \langle Do,\ sta'\rangle \rightarrow_{Do} \langle sta''\rangle \\ \langle Dv,\ var,\ sto\rangle \rightarrow_{Dv} \langle var',\ sto'\rangle \\ var', sto', pri', s \vdash \langle Dr,\ sta'',\ rel\rangle \rightarrow_{Dr} \langle sta''',\ rel'\rangle \\ var', sto', pri', s \vdash \langle Ds_1,\ sto',\ sta'''\rangle \rightarrow_{Ds} \langle sto'',\ sta''''\rangle \\ var, sto, pri, pare_s \vdash \langle Ds_2,\ sto'',\ sta''''\rangle \rightarrow_{Ds} \langle sto''',\ sta'''''\rangle \end{array}}{\begin{array}{r} var, sto, pri, pare_s \vdash \langle \textbf{state}\ s\ \{\ Dp\ Do\ Dv\ Dr\ Ds_1\ \}\ Ds_2,\ sto,\ sta\rangle \rightarrow_{Ds} \langle sto''', sta'''''\rangle \\ \textbf{where}\ (p_r, p_o, p_p, p_c) = sta(pare_s)\ \textbf{and}\ sta' = sta[s \mapsto (p_r, \varepsilon, pare_s, \varepsilon)] \end{array}}$$

(DEC-STATE-EMPTY)
$$var, sto, pri, pare_s \vdash \langle \varepsilon,\ sto,\ sta\rangle \rightarrow_{Ds} \langle sto,\ sta\rangle$$

A program can have an arbitrary amount of States, however the root of the program exists within a global state which is predefined.

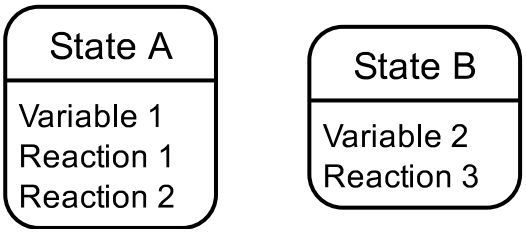

Figure 3.3.: A pair of example states

As shown in fig. 3.3 a State consists of a set of variables and a set of Reactions. These Variables and Reactions are a subset of all the Reactions and Variables in the program, which can be shown as the following:

(3.16)
$$\begin{aligned} AllReactions &= \{R_1, R_2, R_3\} \\ AllVariables &= \{V_1, V_2\} \\ StateA &= \{V_1, R_1, R_2\} \\ StateB &= \{V_2, R_3\} \end{aligned}$$

Where R_1, R_2 and R_3 are reactions and V_1 and V_2 are variables.

A Rios program can change State during runtime. This occurs via Reactions in a State that can cause a Transition. When a State change occurs, the program will complete the

current cycle of Reactions, before checking the new active Reactions in the new State. if a State changes to itself, it will not run any OnEnter operations but will continue from where it stopped.

Priority list

$$
\begin{aligned}
Decs &::= \{Dec\} \\
Dec &::= VarDec \\
&\mid ReactDec \\
&\mid StateDec \\
&\mid OnEnter \\
&\mid PriorityList \\
PriorityList &::= \textbf{priority}\ StateId\ \{\texttt{,}\ StateId\} \\
StateId &::= bigLetter\ \{anyFollowing\}
\end{aligned}
$$

A priority list is a mandatory property of a state. The priority list is meant to make transition collisions deterministic. The priority list for the global state must include all child states, but a child states priority list does not need to include all of its children. This is due to priority lists being inherited through states, where a child will only overwrite the states it sets priority for. This is explained further in section 3.2.10. The following is the operational semantics for priority list declarations:

(DEC-PRIORITY-LIST)

$$\frac{var, rel, pri, pare_s \vdash \langle Dp,\ pri\rangle \rightarrow_{Dp} \langle pri'\rangle \quad \langle s,\ pri'\rangle \rightarrow_{Dp} \langle pri''\rangle}{var, rel, pri, pare_s \vdash \langle s\texttt{,}\ Dp,\ pri\rangle \rightarrow_{Dp} \langle pri''\rangle}$$

(DEC-PRIORITY-ELEMENT)

$$var, rel, pri, pare_s \vdash \langle s,\ pri\rangle \rightarrow_{Dp} \langle pri[s \mapsto p][next \mapsto p+1]\rangle \quad \textbf{where}\ p = pri(next)$$

When determining the final transition in a collision, the priority list is checked. If the current transition has the higher priority, the new transition is discarded. If the new transition has the higher priority, the current transition is replaced by the new transition. The following is the operational semantics for transition discarding and replacement:

(LOCAL-STMT-TRANSITION-REPLACE)

$$\langle \textbf{enter(}\ s\ \textbf{)}, var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto,\ s\rangle \quad \textbf{where}\ pri(s) > pri(next_s)$$

(LOCAL-STMT-TRANSITION-SKIP)

$$\langle \textbf{enter(}\ s\ \textbf{)}, var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto,\ next_s\rangle \quad \textbf{where}\ pri(next_s) > pri(s)$$

Substates

Substates in Rios are States within another State, which, just like normal States, defines which Reactions and Variables are active. Substates in a state is conceptually a seperate automaton, which means it has its own entry point, meaning it can define its own OnEnter or Priority List. When an active state has this sub-automaton, it means one of the states in this automaton will also be active. Because of this, both the active superstates and the active reactions and variables of the substate are active. This is conceptually the same as only the bottom-most state being active, with that set of active reactions of the state, also containing the reactions of the parent state. This can be defined as follows:

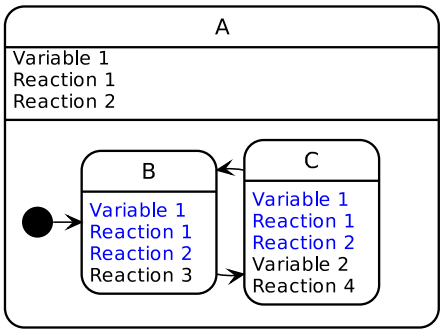

Figure 3.4.: Substating and the relationship between State A, B and C.

As seen in fig. 3.4, Reactions and Variables active within a State are a subset of the total available set of Reactions and Variables in the program. In the case of fig. 3.4 the elements of the global Reactions set range from Reaction 1 - 4. State B is a Substate of State A and therefore the Reactions defined in A is also active when B is the active State. This can be described as:

$$
\begin{aligned}
AllReactions &= \{R_1, R_2, R_3, R_4\} \\
AllVariables &= \{V_1, V_2\} \\
A &= \{V_1, R_1, R_2\} \\
B &= A \cup \{R_3\} \\
C &= A \cup \{V_2, R_4\}
\end{aligned}
\tag{3.17}
$$

A special type of state exists in Rios, called a default state, which attempts to simulate the behavior of the start state mentioned in the I/O automaton thoery. During transitions, if the transition ends in a state with substates which have been marked as default, the transition will continue to that state. This will occur continuously until there are no

substates. This is the same as a start state in an I/O automaton.

(DEC-STATE-DEFAULT)

$$\frac{var, sto, pri, pare_s \vdash \langle \texttt{state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds_1\ \texttt{\}}\ Ds_2,\ sto,\ sta' \rangle \rightarrow_{Ds} \langle sto', sta'' \rangle}{var, sto, pri, pare_s \vdash \langle \texttt{default state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds_1\ \texttt{\}}\ Ds_2,\ sto,\ sta \rangle \rightarrow_{Ds} \langle sto', sta'' \rangle}$$

where $(p_r, p_o, p_p, p_c) = sta(pare_s)$ **and** $p_c = \varepsilon$ **and** $sta' = sta[pare_s \mapsto (p_r, p_o, p_p, s)]$

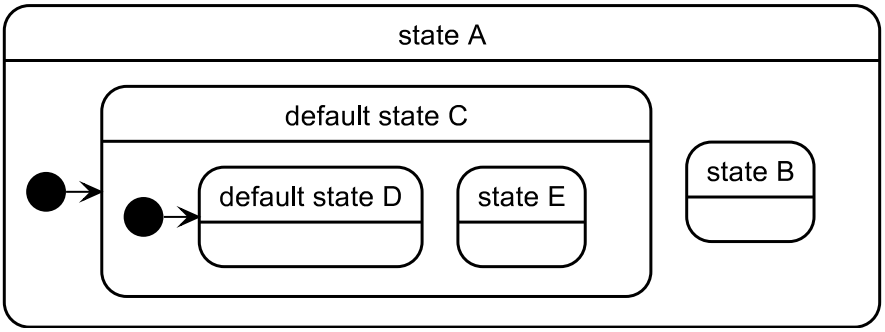

Figure 3.5.: An example state diagram. The diagram illustrates that one can

As an example, consider a state hierarchy as shown in fig. 3.5. If a transition is made to state A, the transition will continue through the default states, until ending in state D, triggering all relevant OnEnter statement sequences. If a transition is made to state B or state E, the transition ends. Likewise, if D had not been the default, it would not have been possible to transition to C without specifying whether D or E is the default.

OnEnter

$$\begin{aligned}
Decs &::= \{Dec\} \\
Dec &::= VarDec \\
&\mid ReactDec \\
&\mid StateDec \\
&\mid OnEnter \\
&\mid PriorityList \\
OnEnter &::= \texttt{onenter}\ Result \\
Result &::= \texttt{:}\ Stmts
\end{aligned}$$

An OnEnter is a property of a state, allowing the state to execute a sequence of statements

before its reactions are evaluated.

(DEC-ONENTER-EMPTY) $$s \vdash \langle \varepsilon, sta \rangle \rightarrow_{Do} \langle sta \rangle$$

(DEC-ONENTER)
$$s \vdash \langle \textbf{onenter :}\ S, sta \rangle \rightarrow_{Do} \langle sta' \rangle \quad \textbf{where}\ sta' = sta[s \mapsto (r^*, S, pare_s, child_s)] \quad \textbf{and}\ (r^*, \varepsilon, pare_s, child_s) = sta(s)$$

The OnEnter is triggered upon a transition to the corresponding state of the OnEnter. It is possible to run multiple OnEnter statement sequences off of a single transition, as seen in fig. 3.5. In this case, the OnEnter that are run are found by taking the set consisting of the next state and its ancestors, and taking its complement with the set of the current state and its ancestors. Both of these sets should be totally ordered, where element $e_1 < e_2$ if e_1 is the child of e_2. The resulting set is the set of states whose OnEnter statement sequences should be run. Formally this can be given as,

$$OnEnter_{current} = \{C, E, G\}$$
$$OnEnter_{next} = \{C, F, H\}$$
$$OnEnter_{run} = OnEnter_{next} \backslash OnEnter_{current} = \{F, H\}$$

Meaning, the OnEnter statement sequences to run are the OnEnter defined within state F and H, in that order. This is also visually represented in fig. 3.6,

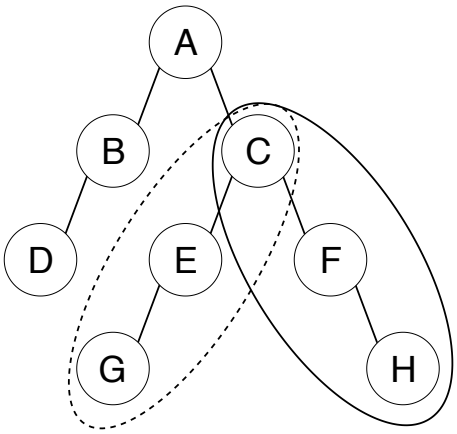

Figure 3.6.: How Rios determines which OnEnter statements to run

The following is the operational semantics for executing OnEnter:

$$\text{(TRANS-TOP)}\qquad \frac{\langle S, var,\ sto, next_s\rangle \rightarrow_l \langle var', sto', next'_s\rangle}{curr_s \vdash \langle s, sto, next_s\rangle \rightarrow_t \langle sto', next'_s\rangle}$$

$$\textbf{where } (-,-,pare_s,-) = sta(s) \textbf{ and } pare_s \in anc(curr_s)$$

$$\text{(TRANS-TRAVERSING)}\qquad \frac{curr_s \vdash \langle pare_s, sto, next_s\rangle \rightarrow_t \langle sto', next'_s\rangle \quad \langle S, var, sto', next'_s\rangle \rightarrow_l \langle var, sto'', next''_s\rangle}{curr_s, next_s \vdash \langle s, sto, next_s\rangle \rightarrow_t \langle sto'', next''_s\rangle}$$

$$\textbf{where } (-, S, pare_s, -) = sta(next_s) \textbf{ and } pare_s \notin anc(curr_s)$$

3.2.3. Reaction

$$\begin{aligned}
Decs &::= \{Dec\}\\
Dec &::= VarDec\\
&\mid ReactDec\\
&\mid StateDec\\
&\mid OnEnter\\
&\mid PriorityList\\
ReactDec &::= When\\
&\mid Always\\
&\mid Every\\
When &::= \texttt{when}\ ExprNum\ (Result \mid [(CompOp|BoolOp)]\ Cases)\\
Always &::= \texttt{always}\ Result\\
Every &::= \texttt{every}\ IntLiteral\ Unit\ (Result \mid When)\\
Cases &::= Case\ \{Case\}\\
Case &::= \texttt{|}\ [(CompOp|BoolOp)]\ Expr\ Result\\
Result &::= \texttt{:}\ Stmts
\end{aligned}$$

A Reaction is an combination of the action types highlighted within the I/O automaton theory, see 3.1.3.

$$\text{(DEC-REACTION)}\qquad \frac{\langle c,\ rel\rangle \rightarrow_{rel} \langle rel'\rangle \quad var, sto, pri, s \vdash \langle Dr,\ sta',\ rel'\rangle \rightarrow_{Dr} \langle sta'',\ rel''\rangle}{var, sto, pri, s \vdash \langle \textbf{when}\ c\ Dr,\ sta,\ rel\rangle \rightarrow_{Dr} \langle sta'', rel''\rangle}$$

$$\textbf{where } sta' = sta[s \mapsto sta(s) \cup \{(c,\ var)\}]$$

$$\text{(DEC-REACTION-EMPTY)}\qquad var, sto, pri, s \vdash \langle \varepsilon, sta, rel\rangle \rightarrow_{Dr} \langle sta, rel\rangle$$

Due to the nature of those action types the Reaction requires an input, either given externally or internally which will satisfy its condition.

(GLOBAL-REACTION)

$$\frac{\langle c,\ var',\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle}{sta,\ rea \vdash \langle\{..., r, ...\}, sto, curr_s, next_s\rangle \Rightarrow_g \langle\{..., ...\}, sto', curr_s, next'_s\rangle} \quad \textbf{where } (c,\ var',\ pri) = r$$

When a the condition of a Reaction is satisfied the corresponding user-specified case will be executed.

(DEC-CAS)
$$\frac{\langle S, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle \quad \langle c, var, rel'\rangle \rightarrow_{rel} \langle var, rel''\rangle}{\langle e : S\ c, var, rel\rangle \rightarrow_{rel} \langle var, rel''\rangle}$$

(DEC-CAS-EMPTY)
$$\langle \varepsilon, var, rel\rangle \rightarrow_{rel} \langle var, rel\rangle$$

The execution of the case is always done in an imperative manner. The expression in each active Reaction is continuously and perpetually evaluated while a program runs, conceptually reacting immediately upon the condition evaluating to true. This implies the possibility of needing to execute several Reactions simultaneously, which could potentially lead to conflicts over the order and manner to execute the Reactions.

(LOCAL-EMPTY)
$$\langle \varepsilon,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto,\ next_s\rangle$$

(LOCAL-CASE-TRUE)
$$\frac{\langle S,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle}{\langle e : S\ c,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle} \quad \textbf{where } e \rightarrow_e true$$

(LOCAL-CASE-FALSE)
$$\frac{\langle c,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle}{\langle e : S\ c,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle} \quad \textbf{where } e \rightarrow_e false$$

3.2.4. Reaction Collision

Since Reactions conceptually run at the same time, ambiguity could arise in certain situations.

```
1    x starts as 0
2    reaction resulting in print(x)
3    reaction resulting in x = x + 5
```

Figure 3.7.: Pseudo code of a Reaction collision.

In fig. 3.7 two Reactions are shown. Incrementing x by 5 and printing it at the same time, can be somewhat ambiguous as to whether the current value of x, at the time

of incrementing, or the updated value of x, after incrementing, is printed. The above Reactions could be printing either "0 5 10 ..." or "5 10 15 ...". This ambiguity can make it difficult to understand the interactions between and reactions, and if allowed to exist, would make generated programs more unreliable without a specified behaviour priority system.

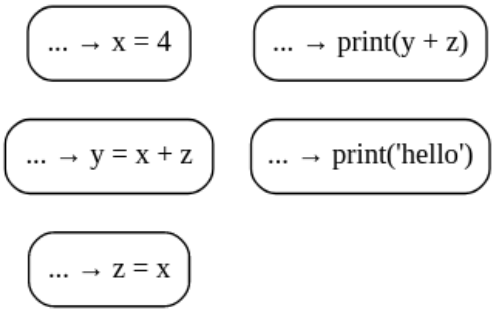

Figure 3.8.: Instructions to run

Another way to illustrate this ambiguity is shown in fig. 3.8. The figure shows five arbitrary Reactions, and some instructions which have to be run after each Reaction has been evaluated. As mentioned earlier these reactions will conceptually run at the same time, but it would cause the program to become non-deterministic. This means that if a read reaction, and a write reaction were to be executed, the compiler would not be able to decide which to choose. This is exemplified in fig. 3.7, where the outcome of the instructions will differ depending on which order the instructions are executed. One way to handle this is by sorting the reactions based on their dependencies.

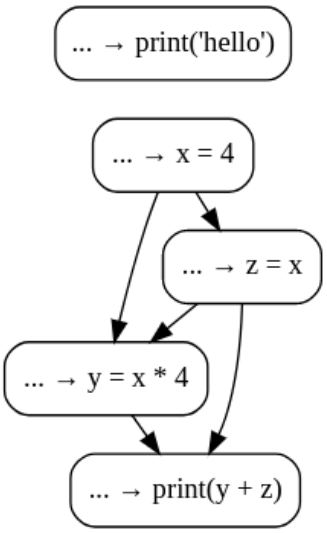

Figure 3.9.: Instructions with dependency ordering shown

Such dependencies can be illustrated with fig. 3.9, where the directions show which way the data flows from supplying Reactions to consuming Reactions. This shows that for these reactions, there are 2 groupings of reactions that can be run simultaneously, as they have no dependence on each other. It does not matter if the Reaction resulting in `print('hello')` is executed before, simultaneously, or after the rest of the instructions. This is essentially thinking of each reaction as a 'rule', where the most independent rules are evaluated before the dependent rules.

However, sorting the Reactions to first evaluate the most dependent rules is also an option. Essentially, this changes the conception behind Reactions from rules, and moves towards having an equivalency in Reactions. Where conceptually all Reactions are evaluated and their respective code is executed in unison. This sorting method ensures that given each 'cycle', a cycle being a single iteration of evaluating all active reactions, all said reactions are evaluated and executed conceptually simultaneously/parallel to each other. The main difference between these two sorting methods is essentially their 'speed'. Given no state transitions, both methods should always execute the same reactions, but at different times. In this regard, each have their benefits, consider the program of an emergency airbag system for a car crash, at fig. 3.10

```
1    //speed is an int and determines the speed of the vehicle
2    //might_be_crashed, engine_broken and crashed are all booleans which
         ↪ describe the state of the vehicle
3    reaction resulting in might-be-crashed = true if speed is 0
4    reaction resulting in crashed = true if engine_broken and
         ↪ might_be_crashed both are true
5    reaction resulting in the activation of airbags if crashed is true
```

Figure 3.10.: Pseudo code of a Reaction collision.

In fig. 3.10 solving the problem by evaluating the independent reactions first is most beneficial. Picture the instant speed of the vehicle reaches 0, then the first reaction will run which allows the second reaction to run and finally allows the last reaction to be executed. However, sorting dependent first, means we have to wait three cycles before reaching the same point. As in the first cycle `'might_be_crashed'` is set to true after the other two reactions have already been evaluated. Then in the second cycle `'crashed'` is set to true, which finally allows the last reaction to run in the beginning of the following cycle. In this case, the speed of executing independent reactions first is valued highly. Now consider another example, shown in fig. 3.11.

```
1   //temperature is an int that causes a state change if it reaches or
        ↪ exceeds a threshold
2   reaction resulting in updating temperature
3   reaction resulting in state change if temperature >= threshold is true
```

Figure 3.11.: Pseudo code of a Reaction collision.

In 3.11 having a slower approach yields a higher accuracy. The cycle where the second Reaction evaluates to true, would then have an additional cycle immediately afterwards where the state transition is executed, and the temperature is updated one final time, ensuring the variable is completely up to date before moving on. In this case, an increased accuracy is desirable. However, it should be mentioned that this same result is also possible by sorting independent first. Imagine if the second reaction also updated the temperature, in this case we would be able to slow down the fast approach, to a more suitable pace. Do note however, that in doing so sacrifices readability in making the program more verbose and reactions multi-purposed. Because of this versatility giving by sorting independent Reactions first, Rios (by default) runs Reactions that declare or change variables, before the Reactions that would use those variables.

The problem regarding the Reaction collisions still exists if there is more than one Reaction which changes the same variable. To avoid this, changing a variable in multiple Reactions in a single state is not permitted in Rios, if the order cannot be implied.

The following are the operational semantics regarding reactions:

(GLOBAL-STARTER)

$$\frac{var, rel, pri \vdash \langle Ds, sto, sta\rangle \rightarrow_{Ds} \langle sto', sta'\rangle}{sta,\ rea \vdash \langle Ds\rangle \Rightarrow_g \langle \emptyset, sto', \texttt{Global}, \varepsilon\rangle} \quad \textbf{where } var, sto, rel, pri, sta = \emptyset \rightharpoonup \emptyset$$

(GLOBAL-REACTION)

$$\frac{\langle c,\ var',\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle}{sta,\ rea \vdash \langle \{..., r, ...\}, sto, curr_s, next_s\rangle \Rightarrow_g \langle \{..., ...\}, sto', curr_s, next'_s\rangle} \quad \textbf{where } (c,\ var',\ pri) = r$$

(GLOBAL-EMPTY-TRANSITION)

$$\frac{curr_s \vdash \langle next_s,\ sto, \varepsilon\rangle \rightarrow_t \langle sto', next'_s\rangle}{sta,\ rea \vdash \langle \emptyset, sto, curr_s, next_s\rangle \Rightarrow_g \langle \{r_1,\ r_2\ , ..., r_i\}, sto', next_s, next'_s\rangle} \quad \textbf{where } (\{r_1,\ r_2\ , ..., r_i\}, S, pare_s, child_s) = sta(next_s) \textbf{ and } child_s = \varepsilon$$

The above semantics describe how the reaction queue begins, how reactions are run (or not), and how the queue is refilled. Additionally, the rules for moving reactions in the

queue is given as concatenation and insertion rules.

(GLOBAL-MERGE-CONCATENATE)

$$sta,\ rea \vdash \langle \{..., r_1, ..., r_2, ...\}, sto, curr_s, next_s \rangle \Rightarrow_g \langle \{..., r_2; r_1, ...\}, sto, curr_s, next_s \rangle$$
$$\textbf{where} \quad \begin{array}{r} \exists l \in rel(r_2, writer) : l \in rel(r_1, reader) \\ \forall l \in rel(r_1, writer) : l \notin rel(r_2, writer) \cup rel(r_2, reader) \end{array}$$

(GLOBAL-MERGE-INSERT)

$$sta,\ rea \vdash \langle \{..., r_1, ..., r_2; r_3, ...\}, sto, curr_s, next_s \rangle \Rightarrow_g \langle \{..., r_2; r_1; r_3, ...\}, sto, curr_s, next_s \rangle$$
$$\begin{array}{rr} & \exists l \in rel(r_2, writer), l \in rel(r_1, writer) \\ & \exists l \in rel(r_1, writer), l \in rel(r_3, reader) \\ \textbf{where} & \forall l \in rel(r_3, writer), l \notin rel(r_1, reader) \cup rel(r_1, writer) \\ & \forall l \in rel(r_3, writer), l \notin rel(r_2, reader) \cup rel(r_2, writer) \\ & \forall l \in rel(r_1, writer), l \notin rel(r_2, reader) \cup rel(r_2, writer) \end{array}$$

3.2.5. Variables

$$\begin{array}{rcl} Decs & ::= & \{Dec\} \\ Dec & ::= & VarDec \\ & | & ReactDec \\ & | & StateDec \\ & | & OnEnter \\ & | & PriorityList \\ VarDec & ::= & Type\ Id\ Assign\ Expr \\ Type & ::= & \texttt{apin} \mid \texttt{dpin} \mid \texttt{serial} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{byte} \mid \texttt{float} \mid \texttt{long} \\ Id & ::= & smallLetter\ \{anyFollowing\} \\ smallLetter & ::= & \texttt{a} \ldots \texttt{z} \\ anyFollowing & ::= & \texttt{a} \ldots \texttt{z} + \texttt{A} \ldots \texttt{Z} \\ Assign & ::= & \texttt{=} \end{array}$$

In Rios, a Variable is an identifier coupled with some value. Variables can be datatypes like numbers, strings, characters, pins or states. In Rios, Variables are in one of two different scope types, which will be described in section 3.2.5 and section 3.2.5 respectively.

Local Variables

Variables can be declared in reactions. These Variables can only be used in the reaction in which they are declared, and are discarded as soon as the reaction has completed. If

the reaction is run again, the Variable is defined again.

(LOCAL-STMT-VAR-DEC)

$$\frac{var, sto \vdash e \rightarrow_e v}{\langle t\ x\ \texttt{=}\ e,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var',\ sto',\ next_s\rangle}$$

where $l = sto(next)$ **and** $var' = var[x \mapsto l][next \mapsto next(l)]$ **and** $sto' = sto[l \mapsto v]$

State Variables

Variables can also be declared in a state. The Variables declared in the state can be used by all reactions contained within the state, including reactions in substates. These Variables live for all of the programs duration, so when the state is exited and then reentered, the variables still maps to the old value. As Rios uses static scoping, Variables in a substate can have the same name as Variables a parent State, but they will be treated as different entities.

The following are the operational semantics regarding variables:

(VAR-REF) $\quad var, sto \vdash var,\ sto \vdash x \rightarrow_e v \quad$ **where** $v = sto(var(x))$

(DEC-VAR)

$$\frac{var, sto \vdash \langle e,\ var,\ sto\rangle \rightarrow_e v \quad \langle Dv,\ var,\ sto\rangle \rightarrow_{Dv} \langle var',\ sto'\rangle}{\langle t\ x\ \texttt{=}\ e\ Dv,\ var,\ sto\rangle \rightarrow_{Dv} \langle var'[x \mapsto l],\ sto'[l \mapsto v][next \mapsto next(l)]\rangle} \quad \textbf{where } l = var[next]$$

(DEC-VAR-EMPTY) $\quad \langle \varepsilon,\ var,\ sto\rangle \rightarrow_{Dv} \langle var,\ sto\rangle$

(DEC-STMT-VAR)

$$\frac{\langle e, var, rel\rangle \rightarrow_{rel} \langle var', rel'\rangle}{\langle t\ x\ \texttt{=}\ e, var, rel\rangle \rightarrow_{rel} \langle var'[x \mapsto var(next)], rel'\rangle}$$

(DEC-EXPR-VAR) $\quad \langle x, var, rel\rangle \rightarrow_{rel} \langle var, rel[(x, reader) \mapsto rel(x, reader) \cup \{r\}]\rangle$

3.2.6. Statements

$$
\begin{aligned}
Stmts &::= \texttt{[}\ Stmt\ \{\texttt{;}\ [Stmt]\}\ \texttt{]} \\
&\mid Stmt \\
Stmt &::= VarDec \\
&\mid TransitionStmt \\
&\mid Expr\ [\{\texttt{,}\ Expr\}\ (CompAssign|Assign)\ Expr] \\
TransitionStmt &::= \texttt{transition (}\ StateId\ \texttt{)} \\
Calls &::= \texttt{read} \\
&\mid \texttt{toggle} \\
&\mid \texttt{write} \\
&\mid \texttt{println}
\end{aligned}
$$

Statements are imperatively executed instructions in a program in Rios. Statements can be chained together in sequences, with each statement being separated by semicolons. A sequence of statements is always encapsulated in a pair of square brackets.

(DEC-STMT-SEQ)
$$\frac{\langle S_1, var, rel\rangle \rightarrow_{rel} \langle var', rel'\rangle \quad \langle S_2, var', rel'\rangle \rightarrow_{rel} \langle var'', rel''\rangle}{\langle S_1\texttt{;}\ S_2, var, rel\rangle \rightarrow_{rel} \langle var'', rel''\rangle}$$

(LOCAL-STMT-SEQUENCE)
$$\frac{\langle S_1,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var',\ sto',\ next'_s\rangle \quad \langle S_2,\ var',\ sto',\ next'_s\rangle \rightarrow_l \langle var'',\ sto'',\ ''\rangle}{\langle\ S_1\texttt{;}\ S_2,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var'',\ sto'',\ ''\rangle}$$

Statements are semantically instructions, telling what operation to execute. Transition instructions are covered in section 3.2.8, and variable declaration instructions are covered section 3.2.5.

(LOCAL-STMT-ASS)
$$\langle x\ \texttt{=}\ e\ ,\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto[l \mapsto v],\ next_s\rangle \quad \textbf{where } var, sto \vdash e \rightarrow_e v \textbf{ and } l = var(x)$$

(DEC-STMT-IGNORE)
$$\langle S, var, rel\rangle \rightarrow_{rel} \langle var, rel\rangle \quad \textbf{where } S \in \{\varepsilon,\ \texttt{transition(}\ s\ \texttt{)}\}$$

(DEC-STMT-ASS)
$$\frac{\langle e, var, rel\rangle \rightarrow_{rel} \langle var', rel'\rangle}{\langle x\ \texttt{=}\ e, var, rel\rangle \rightarrow_{rel} \langle var', rel'[(r, writer) \mapsto rel(r, writer) \cup \{l\}]\rangle} \quad \textbf{where } l = var(x)$$

(3.18)

3.2.7. Expressions

$$
\begin{aligned}
\mathit{Expr} &::= \mathit{ExprBool} \\
\mathit{ExprBool} &::= \mathit{ExprComp}\ [\mathit{BoolOp}\ \mathit{ExprBool}] \\
\mathit{ExprComp} &::= \mathit{ExprNum}\ [\mathit{CompOp}\ \mathit{ExprComp}] \\
\mathit{ExprNum} &::= \mathit{ExprProduct}\ [\mathit{NumOp1}\ \mathit{ExprNum}] \\
\mathit{ExprProduct} &::= \mathit{ExprNegate}\ [\mathit{NumOp2}\ \mathit{ExprProduct}] \\
\mathit{ExprNegate} &::= [\texttt{!}]\ \mathit{ExprCall} \\
\mathit{ExprCall} &::= \mathit{ExprSingle}\ [[\texttt{.}\ \mathit{Calls}]\ \texttt{(}\ [\mathit{Expr}\ \{\texttt{,}\ \mathit{Expr}\}]\ \texttt{)}] \\
\mathit{ExprSingle} &::= \mathit{Literal} \\
&\quad | \ \mathit{Id} \\
&\quad | \ \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
\mathit{Literal} &::= \mathit{IntLiteral} \\
&\quad | \ \mathit{ByteLiteral} \\
&\quad | \ \mathit{LongLiteral} \\
&\quad | \ \mathit{FloatLiteral} \\
&\quad | \ \mathit{PinLiteral} \\
&\quad | \ \mathit{SerialLiteral} \\
&\quad | \ \mathit{StringLiteral} \\
&\quad | \ \mathit{BoolLiteral}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Calls} &::= \texttt{read} \\
&\quad | \ \texttt{toggle} \\
&\quad | \ \texttt{write} \\
&\quad | \ \texttt{println}
\end{aligned}
$$

An expression is a legal statement in the Rios language, such as the ability to negate a statement or compare two same typed variables with each other, except values of the void type.

(DEC-EXPR-BIN) $$\frac{\langle e_1, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle \quad \langle e_1, var, rel'\rangle \rightarrow_{rel} \langle var, rel''\rangle}{\langle e_1\ op\ e_2, var, rel\rangle \rightarrow_{rel} \langle var, rel''\rangle}$$
where $op \in \{$`+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `||`, `&&`$\}$

(DEC-EXPR-SIN) $$\frac{\langle e, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle}{\langle \texttt{!}e, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle}$$

(DEC-EXPR-NUM) $$\langle n, var, rel\rangle \rightarrow_{rel} \langle var, rel\rangle$$

A Single expression type is the natural value found. As such, no changes are made to the actual expression and no operators are applied.

(EXPT-NUMERAL) $var, sto \vdash var,\ sto \vdash n \rightarrow_e v$ **where** $\mathcal{N}(n) = v$

(VAR-REF) $var, sto \vdash var,\ sto \vdash x \rightarrow_e v$ **where** $v = sto(var(x))$

(EXPR-PARENTHESIS) $var, sto \vdash$ `(` e `)` $\rightarrow_e v$

A call is an operation that is accessed by adding a postfix `.` and then one of the four call operations. All of the call operations require a parameter, and then return a value of the type void. Naturally, **`read`** is excluded from this rule, as it simply returns the written value on the specified pin. The **`read`** and **`write`** calls can be called on any pin, digital or analogue.

(EXPR-PIN-READ)
$var, sto \vdash x$`.read()` $\rightarrow_e v$ **where** $v =$ the value of the pin at $sto(var(x))$

(EXPR-PIN-WRITE) $$\frac{var, sto \vdash e \rightarrow_e v_1}{var, sto \vdash x\texttt{.write(}\ e\ \texttt{)} \rightarrow_e v_2}$$
where The pin is set to v **and** $v_2 =$ *void*

On the contrary, **`toggle`** can only be used on a dpin, and **`println`** can not be applied to a pin at all. The **`toggle`** call can be used on a digital pin or a Boolean variable. The call will negate the value of the given pin or Boolean, setting it high if it is low or vice versa. The **`println`** attempts to print a given parameter.

(EXPR-PIN-TOGGLE)
$var, sto \vdash x$`.toggle` $\rightarrow_e v$
where The pin is set to the opposite possible value **and** $v =$ *void*

(EXPR-SERIAL-PRINTLN) $$\frac{var, sto \vdash e \rightarrow_e v_1}{var, sto \vdash x\texttt{.println(}\ e\ \texttt{)} \rightarrow_e v_2}$$
where Write v_1 to the serial connection at $sto(var(x))$ **and** $v_2 =$ *void*

Given that calls are pin type dependent, implies the valid values for an analogue pin and a digital pin are not the same. An analogue pin will accept a write of, and return a read of the type int. A digital pin will accept a write of as well as a toggle, and return a read of the type bool.

Any expression type may be inverted at any time, this utilises the Negate expression type. The inversion seen here is similar to that in mathematics.

(EXPR-NEGATE-TRUE) $\quad var, sto \vdash \texttt{!}e \rightarrow_e true \quad \textbf{where } e \rightarrow_e false$

(EXPR-NEGATE-FALSE) $\quad var, sto \vdash \texttt{!}e \rightarrow_e false \quad \textbf{where } e \rightarrow_e true$

On the other hand, expression types such as Num and Product mathematically simplifies an expression, returning the simplified product of the respective operation. These expression types covers all available operations found within elementary arithmetic, and follow the order of operations found within mathematics.

(EXPR-ADD) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ + } e_2 \rangle \rightarrow_e v} \quad \textbf{where } v = v_1 + v_2$$

(EXPR-SUB) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ - } e_2 \rangle \rightarrow_e v} \quad \textbf{where } v = v_1 - v_2$$

(EXPR-MULT) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ * } e_2 \rangle \rightarrow_e v} \quad \textbf{where } v = v_1 \cdot v_2$$

(EXPR-DIV) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ / } e_2 \rangle \rightarrow_e v} \quad \textbf{where } v = \frac{v_1}{v_2}$$

The Comp expression type attempts to test whether a relation given by a comparative operator is true or false.

(EXPR-EQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ == } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 = v_2$$

(EXPR-EQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ == } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 \neq v_2$$

(EXPR-NOTEQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ != } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 \neq v_2$$

(EXPR-NOTEQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ != } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 = v_2$$

(EXPR-LESSTHAN-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ < } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 < v_2$$

(EXPR-LESSTHAN-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ < } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 > v_2$$

(EXPR-BIGGERTHAN-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ > } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 > v_2$$

(EXPR-BIGGERTHAN-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ > } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 < v_2$$

(EXPR-LESSTHANOREQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ <= } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 = v_2 \vee v_1 < v_2$$

(EXPR-LESSTHANOREQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ <= } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 > v_2$$

(EXPR-GREATERTHANOREQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ >= } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 = v_2 \vee v_1 > v_2$$

(EXPR-GREATERTHANOREQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ >= } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 < v_2$$

The final expression type is Bool type, which may extend the number of premises necessary to reach a conclusion. It tests whether the relationship between two states can be expressed with a given Boolean operator.

(EXPR-BOOLAND-TRUE)

$$var, sto \vdash e_1 \texttt{ \&\& } e_2 \rightarrow_e true \quad \textbf{where } e_1 \rightarrow_e true \ \wedge e_2 \rightarrow_e true$$

(EXPR-BOOLAND-FALSE)

$$var, sto \vdash e_1 \texttt{ \&\& } e_2 \rightarrow_e false \quad \textbf{where } e_1 \rightarrow_e false \ \vee e_2 \rightarrow_e false$$

(EXPR-BOOLOR-TRUE)

$$var, sto \vdash e_1 \texttt{ || } e_2 \rightarrow_e true \quad \textbf{where } e_1 \rightarrow_e true \ \vee e_2 \rightarrow_e true$$

(EXPR-BOOLOR-FALSE)

$$var, sto \vdash e_1 \texttt{ || } e_2 \rightarrow_e false \quad \textbf{where } e_1 \rightarrow_e false \ \wedge e_2 \rightarrow_e false$$

Operator Precedence Rules

While the precedence rules for Rios can be extracted from the specified CFG, a simplified table is presented in 3.2.7 which shows all operator precedents. The table consists of four parts: The priority, where the higher the number the higher priority; Operator and description, which shows the operator and a relevant description for the operator in question; Associativity, which determines the order of processing should an expression contain operators with the same precedence.

Precedence	Operator	Description	Associativity
1	`()`	Subexpression	Left to Right
2	`()`	Calls	Left to Right
3	`!`	Logical negation	Left to Right
4	`*`	Multiplication	Left to Right
	`/`	Division	
5	`+`	Addition	Left to Right
	`-`	Substraction	
6	`==`	Equality	Left to Right
	`!=`	Inequality	
	`<=`	Less than or equal to	
	`>=`	Greather than or equal to	
	`<`	Less than	
	`>`	Greater than	
7	`&&`	Logical AND	Left to Right
	`||`	Logical OR	
8	`=`	Assignment	Left to Right

Table 3.1.: Overview of operator precedence rules.

3.2.8. Transition

$$
\begin{aligned}
Stmts &::= \texttt{[}\ Stmt\ \{\texttt{;}\ [Stmt]\}\ \texttt{]} \\
&\mid Stmt \\
Stmt &::= VarDec \\
&\mid TransitionStmt \\
&\mid Expr\ [\{\texttt{,}\ Expr\}\ (CompAssign|Assign)\ Expr] \\
TransitionStmt &::= \texttt{transition (}\ StateId\ \texttt{)}
\end{aligned}
$$

Transitions in Rios is a way to simulate a transition relation (s', π, s), meaning that for every state `s'` and input action π there is a transition resulting in `s`.

(LOCAL-STMT-TRANSITION-FIRST) $\langle \texttt{enter(}\ s\ \texttt{)},\ var,\ sto,\ \varepsilon\rangle \rightarrow_l \langle var,\ sto,\ s\rangle$

(LOCAL-STMT-TRANSITION-REPLACE)
$\langle \texttt{enter(}\ s\ \texttt{)},\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto,\ s\rangle$ **where** $pri(s) > pri(next_s)$

(LOCAL-STMT-TRANSITION-SKIP)
$\langle \texttt{enter(}\ s\ \texttt{)},\ var,\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto,\ next_s\rangle$ **where** $pri(next_s) > pri(s)$

Given that Rios does not make use of input actions as a concept, but rather as part of the greater idea of reactions, transitions are exclusively limited to reactions simulating the behaviour of an input action. However, because of the versatility of the reaction and its ability to limit itself to replicate an input action, it is possible to program a language in Rios which only relies on input actions to cause transitions, therefore simulating an IO automata. From a semantic viewpoint, this is handled via a reaction-queue. This queue works by taking a set of reactions held within a state and adding them to the queue as explained in section 3.2.8. When the program encounters an enter command it queues the transition, resolves the remaining reactions within the reaction queue, and fills the reaction-queue with the contents of the new state and its associated states.

Transition Collision

In some cases, a program can encounter multiple actions that demand the program to jump to different states. This actions is referred to as a transition-collision, and has a specific set of semantic rules to solve such problems. A Transition collision occurs when multiple Reactions wish to transition to different States in the same cycle. The Transition-collisions cannot be handled in the same manner as the Reaction-collisions due to the lack of dependencies. Therefore, making a inferred ordering of the state changes is not an option.

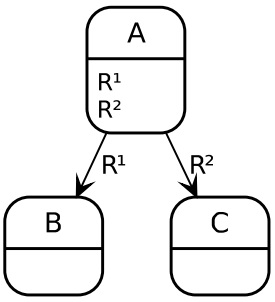

Figure 3.12.: A case where transitions can collide.

An example of this is illustrated in fig. 3.12, where both R^1 and R^2 could evaluate to true. Rios solves which to pick through an implicit ordering priority, or by a user-defined priority list. The priority list must include a number of states, placed on the list in a sequential fashion, where the first entry is the most important, and the last entry is the least. For states not in the list, or in the case of no list being defined, a compiler error will be issued instead.

The operational semantics for transitions is as follows:

(GLOBAL-EMPTY-TRAVERSE)

$$sta,\ rea \vdash \langle \emptyset, sto, curr_s, next_s \rangle \Rightarrow_g \langle \emptyset, sto, curr_s, next'_s \rangle$$
$$\textbf{where } (-,-,-,next'_s) = sta(curr_s) \textbf{ and } curr'_s \neq \varepsilon$$

(GLOBAL-EMPTY-TRANSITION)

$$\frac{curr_s \vdash \langle next_s,\ sto, \varepsilon \rangle \rightarrow_t \langle sto', next'_s \rangle}{sta,\ rea \vdash \langle \emptyset, sto, curr_s, next_s \rangle \Rightarrow_g \langle \{r_1,\ r_2\ , ..., r_i\}, sto', next_s, next'_s \rangle}$$
$$\textbf{where } (\{r_1,\ r_2\ , ..., r_i\}, S, pare_s, child_s) = sta(next_s) \textbf{ and } child_s = \varepsilon$$

3.2.9. Types

$$
\begin{aligned}
\mathit{Unit} &::= \texttt{d} \mid \texttt{h} \mid \texttt{m} \mid \texttt{s} \mid \texttt{ds} \mid \texttt{cs} \mid \texttt{ms} \\
\mathit{Type} &::= \texttt{apin} \mid \texttt{dpin} \mid \texttt{serial} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{byte} \mid \texttt{float} \mid \texttt{long} \\
\mathit{BoolLiteral} &::= \texttt{true} \mid \texttt{false} \mid \texttt{high} \mid \texttt{low} \\
\mathit{PinLiteral} &::= (\texttt{a} \mid \texttt{analog} \mid \texttt{d} \mid \texttt{digital})\ \texttt{@}\ [\texttt{A}]\ \mathit{IntLiteral}\ [\texttt{pullup}] \\
\mathit{SerialLiteral} &::= \texttt{\$}\ (\texttt{usb} \mid \texttt{pin})\ \mathit{IntLiteral} \\
\mathit{StringLiteral} &::= \texttt{"}\ \{\mathit{notQuote}\}\ \texttt{"} \\
\mathit{IntLiteral} &::= \mathit{digit}\ \{\mathit{digit}\} \\
\mathit{LongLiteral} &::= \mathit{digit}\ \{\mathit{digit}\}\ \texttt{L} \\
\mathit{ByteLiteral} &::= \mathit{digit}\ \{\mathit{digit}\}\ \texttt{B} \\
\mathit{FloatLiteral} &::= \mathit{digit}\ \{\mathit{digit}\}\ \texttt{.}\ [\mathit{digit}\ \{\mathit{digit}\}] \\
\mathit{digit} &::= \texttt{0} \ldots \texttt{9}
\end{aligned}
$$

Byte, int, & long

A byte, int or long is an integer $n \in \mathbb{Z}$. Byte is represented by the smallest data on a given platform, usually assumed to be a byte of 8 bits. Int is represented is given by a platform's word size, while a long is given by a platform's double-word size.

As an example, a platform has a byte size of 8 bits, a word size of 16 bits and a double-word size of 32 bits. On this platform a byte can contain values ranging from -2^{8-1} to $2^{8-1}-1$ or -128 to 127. An int can contain values ranging from -2^{16-1} to $2^{16-1}-1$ or -32.768 to 32.767, and a long can contain values ranging from -2^{32-1} to $2^{32-1}-1$ or $-2.147.483.648$ to 2.147.483.647

Float

A float is a number $n \in \mathbb{Q}$. Float is represented by the given platforms implementation of float. For example, most arduino platforms implement floats as a 4-byte datatype, giving a range of numbers from $3.4028235 \cdot 10^{38}$ to $-3.4028235 \cdot 10^{38}$. In general, as the value of a float gets further from 0, its accuracy decreases, simply due to how the float is read as binary data.

Apin and Dpin

Rios utilises two custom types revolving around pins found on microcontrollers. Instead of using a basic integer type to define pins and set their pin mode, they are instead given

a pin type that comes with a pre-defined set of properties. An apin is a representation of analogue pins on a given microcontroller platform. As such, the given apin can be used for any purpose an analogue pin can be used for, on a given platform. Likewise, a dpin is a representation of a digital pin, which can be used as any digital pin on a given platform. These pin types serve as the primary means of communicating with the outside world within Rios.

As an example, if there were to occur a situation in which an apin was to be used for turning on an LED. Said pin would be defined using a structure akin to $apinled = a@13$.

Serial

The idea of the serial connection in Rios is reminiscent of the one present in Arduino Language. It is used as the primary communication medium between an Arduino board and a computer or other device like a serial monitor. The major difference from Arduino Language is that a programmer can define more than one serial connection in Rios, enabling programs to deal with multiple devices at the same time directly from the Arduino board - whether it is with other Arduino boards, or computers.

Bool

A bool is a type that represent a boolean value. The reasoning for having two different of ways of writing a bool is due to users from the Arduino Language being more accustomed to high and low, while users from other programming language may find true and false more familiar.

The types rules of Rios can be found in Appendix D

3.2.10. Static Scoping

Rios uses static scoping, applying it to many concepts in the language. Scopes are encapsulated by states and reactions, with reaction scopes being created per case evaluated in the reaction. For example, a reaction with two cases to evaluate can have variable X in one and variable Y in the other. Each variable can be accessed in the own cases, but cannot be accessed from the other.

The reasoning behind picking static over dynamic scoping was two-fold. One reason was that it was decided during the design phase, that Rios should resemble C++ and Arduino Language, both of which have static scoping. The other reason was based in the fact that it was easier to implement with the I/O automata being employed. This is because of the way environments are saved within static scopes. When employing static scopes the variable environment can be stored as is, whereas it is necessary to change the environment every time a change occurs within dynamic scoping.

States and Priority Lists

States is the part of Rios that most resembles C or Java-like scopes. Variables in states can be accessed by reactions and substates contained by the state. This applies recursively for any substates. In the case of a priority list in both a child and parent, a given reaction uses the priority list in its containing scope. Subsequent scopes with new priority lists may specify less states than the parent list, in which case the child priority "overwrites" the specified states. For example, consider two priority lists like so:

$$ParentPriority := A, B, C, D$$
$$ChildPriority := C, A$$

The resulting priority list in the child would be {C, A, B, D}, while the priority list for the parent would remain {A, B, C, D}. A reaction in the parent state would then use the parents priority list, while a reaction in the child state would use the childs priority list.

Reactions

Reactions are not widely affected by static scoping, apart from cases having individual scopes. For example, if a reaction is given as

```
1 when x <
2     | 10 : [int y = 1]
3     | 5  : [int z = 1]
```

then the case on line 2 would not be able to access variable z, and the case on line 3 would not be able to access variable y.

Variables

Variables can only be accessed by their encapsulating scope and any subscopes of it. If a variable of the same name is declared in a subscope, the new variables is used in the following code.

3.3. Summary of Chapter 3

In this chapter, the initial requirements for Rios were presented, as well as how they were fulfilled. Additionally, the core concepts and types that the language implements were also defined, as well as how these work in practice. Finally, was the grammar for the language and the semantic definitions for the different concepts and types in the language.

By reading this chapter, readers should have gained an understanding of the fundamental constructs within Rios, as well as gained an overview of the inter-dependencies that each production has. Additionally, the reader should have insight into the conditions, set forth by the authors, that Rios should be able to fulfil, before being classified as a complete system.

4. The Compiler

When using a language, either for programming or other linguistic purposes, humans use commonly accepted constructs. Likewise, a computer uses syntax to understand which productions to commit to. To go from written words to a semantically equivalent, but entirely different language, or even low-level machine code, it is necessary to create a mapping between the two. This is where a compiler is used.

This chapter describes the process of how the Rios compiler was built. The chapter gives a thorough explanation of the various modules that the Rios compiler consists of, and the associated design choices of said modules.

4.1. Understanding the Compiler

At its core, a compiler converts code in a given programming language to another target language. It does this via multiple lesser modules that each handle a part of the compilation process. There are many ways of assembling a compiler, all dependent on the context it is used in. The basic structure can be seen within Figure 4.1

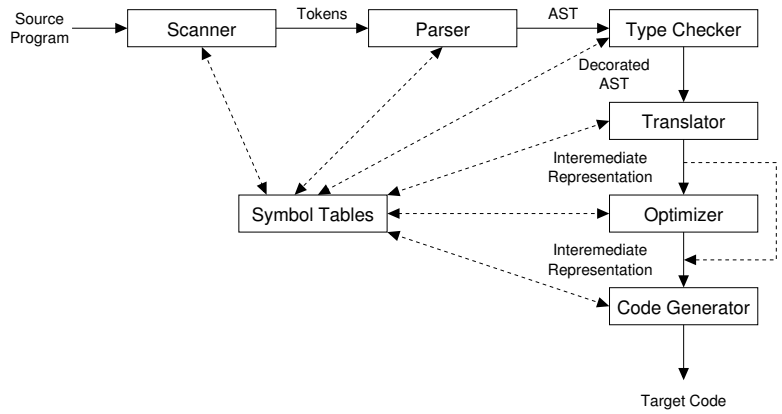

Figure 4.1.: The different modules of a compiler. The example is taken from Crafting a Compiler [16, p. 15].

The scanner, also called the lexer, begins the analysis of a program by reading the input text and grouping individual characters together. It creates a stream of an abstracted

data-type in the form of tokens, based off of these terminal strings. It then eliminates unneeded information like whitespace, and processes compiler directives. In addition to this, the scanner in some compilers will in some cases enter preliminary information into symbol tables, as well as format and list the source program. [16, p. 16]

The parser reads tokens from the scanner and groups them in accordance with a given syntax specification. The parser verifies correct syntax, and if errors are found, issues suitable error messages. In some cases it can try to repair errors or continue execution in spite of the errors. The parser usually builds an Abstract Syntax Tree, which serves as the basis for semantic processing. [16, p. 16-17]

The type checker checks the static semantics of each AST node, by ensuring that the constructed node represents a legal and meaningful construct. If proven to be correct, the type checker will add type information to the node. If this turns out not to be the case, the type checker will instead issue suitable error messages. [16, p. 17]

The translator takes the AST nodes created by the type checker, and turns them into intermediate representation (IR) code, that implements the meaning of the AST nodes. An AST node of an if statement, has no indication as to have to behave. It is not until the translator handles this node, that the notion of testing a value, and conditionally running the statement is created. [16, p. 17]

The optimiser improves upon the IR code by simplifying, moving or removing unneeded components. The optimisation process is often done multiple times, and can take place both before and after the code generation stage. [16, p. 19]

The code generator maps the IR code to machine code for the system that needs to run it. This is done by collecting extensive amounts of information about the target machine, such as register allocation and code scheduling. While the other parts of a compiler can be automatically generated by various tools, the code generator is often created by hand. This is because machine-specific optimisation requires consideration for special cases unique to the target system. [16, p. 19]

Symbol Tables is a mechanism used across the various compiler phases. It allows for information to be associated with identifiers, that can be saved and used across compiler phases. This means that whenever an identifier is declared, the symbol table provides all the information collected about it across all compiler modules. [16, p. 18]

4.1.1. Understanding the Rios Compiler

The model for the Rios compiler differs slightly from the standard model depicted in section 4.1. The reasoning behind the difference is mainly because of the reactions shown in section 3.2. Ideally, Rios should handle all reactions simultaneously, but this is not feasible due to most Arduino boards not allowing multithreading. Because of this, some changes to the compiler modules were made to allow for handling and prioritising said reactions in accordance with rules of determinism within syntax design. The compiler

structure used within Rios can be viewed within Figure 4.2. All of the intricacies of the different phases, and all associated choices, will be explained in depth in subsequent sections, but a rudimentary explanation will first be made here.

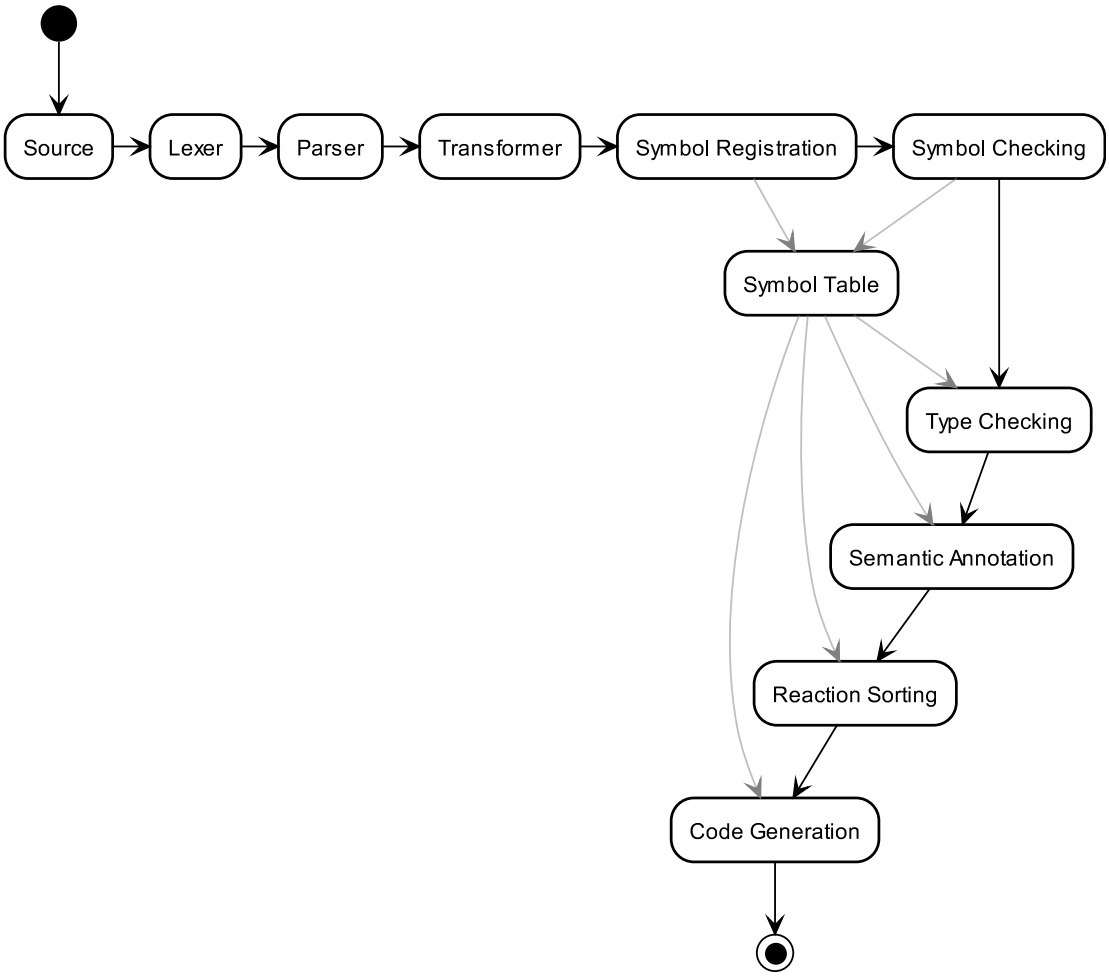

Figure 4.2.: The compiler structure employed within Rios

The Rios lexer works much like the scanner explained in section 4.1. It reads the input text and groups individual characters together into tokens. The only deviation is that it refrains from inserting anything into symbol tables. This is because of the reactions found within the language. Reactions in Rios are dependent on the scope they reside in, and as such, requires that all scopes are created, before being able to insert them into symbol tables. This is instead handled within the symbol registrator and symbol checker.

The Rios parser is designed to do the things explained within the parser in section 4.1. It is modelled to be a recursive-descent parser working on an LL(1) grammar. Other

than this, the parser is mostly ordinary.

The Rios transformer is a module that does two things. First, it creates a global state node, that serves as a wrapper for all the actual states within a given program being compiled. Secondly, it locates different syntactic constructs defined in the language, and changes them to be uniform to each other.

The Rios symbol registrator is a separate module that enters preliminary information into the symbol table. As mentioned in the lexer, this is a function that can normally be done by the scanner. However, because Rios works almost exclusively via states, these states have to be generated before symbols can be checked. The way this works, is that whenever a state has to be checked for symbols, it will be put on the stack, and then checked for new symbols that will be added to the symbol table. That scope will then be popped from the stack, when the symbol registration process for that state is done.

The Rios symbol checker checks the symbol registrations found within a program. It checks that symbols are used as intended and in some cases makes new nodes. These new nodes are references to the original nodes visited. This is done because of the depth-first visitor patterns used within Rios. These new child nodes are used by the transformer when it visits the various scopes. Because the transformer starts at an arbitrary declaration, it might in some cases need to change parental nodes, which is where these new nodes are used, as the old nodes cannot be assigned new parents.

The Rios type checker makes sure that values match the corresponding type declared. There is no coercion within Rios, meaning that types will not be converted to other types to make an expressions valid. As an example, `int` types can only be used with other `int` types, and not `float` types or `bool` types. As such, the type checker validates whether operations are legal, by checking if declared values match their inferred type.

The Rios reaction sorter changes the specific imperative ordering of reactions within states. When a program is run, the reaction sorter runs a topological sort that sorts write actions to variables before read actions of variables. This means that operations that change variables will happen, and then operations that read and act based off of that, will be run second. As a result, reaction sorting is where potential reaction collisions are handled. As the program, on a fundamental level, runs in an imperative fashion, this module makes sure that no two reactions are run at the same time.

The Rios code generator traverses the tree and uses that to create code. Rios compiles down to the Arduino Language as an IR language. From there it goes through the Arduino-pipeline and ends up as low-level machine code.

4.2. The Rios Lexer

The lexer for Rios is simple by design. When called upon, the lexer will analyse the input source code until a token can be created. Once this is done, the token is returned,

and the lexer stops until it is once again caled upon by the parser. How the parser does this, will be further described in section 4.4 and section 4.5. The tokens created contain information about their type (operators, reaction type, parentheses, brackets, etc.) and their value ("==", "[", "variableName", "123", etc.). Tokens also contain info about which line of code, and how far into the line, it occurs. This sequential approach, that handles each part of input strings systematically, makes debugging easy. Because of this approach, the compiler knows exactly where an error occurs, if a user creates bad quote, thus resulting in very precise compiler errors.

4.3. Lexer Generating Tools

There are a number of tools that can generate lexers automatically, when given a CFG. These will analyse the CFG and generate a lexer that can recognise an input conforming to the CFG. The project has considered ANTLR, COCO/R, and manual creation to create a lexer and parser, with the final lexer being made manually. ANTLR and COCO/R will be discussed in section 4.5.1 and section 4.5.2 respectively.

As mentioned previously, the lexer works when called by the parser, creating a token stream as the tokens are needed. An alternative method of lexing stems from the idea of the lexer being run once, creating a list of tokens from the source program as output. The group chose the former method. This meant that instead of creating automaton-like structure, it was opted to pick the always pick a simpler and more maintainable solution, at the cost of some performance.

4.4. Picking a Parsing Approach

The two most widely-used parsing approaches are LL(k) or LR(k), often referred to as top-down and bottom-up parsers respectively [16, p. 126]. LL(k) and LR(k) are abbreviations, indicating how the parser handles input, which kind of parser is created, and how many tokens can be peeked ahead. In both parsing techniques, the first letter stands for *left-to-right*, and indicated that the parser will start at the leftmost token, and move towards the right when processing input. The second letter, is where the parsing techniques vary, and stands for either *leftmost derivation* or *rightmost derivation in reverse*. Leftmost derivation always picks the leftmost nonterminal in every possible situation, adding elements to the parse tree in a depth-first fashion [16, p. 116]. Rightmost derivation, on the other hand, always picks the rightmost nonterminal for any situation. Additionally, it traces the derivations and applies them in reverse order. The last step will be the first element added to the parse tree, and the first step will be the last, much akin to how a topological search is conducted [16, p. 116-117]. Lastly k, also called lookahead, defines how many tokens the parser can peek ahead of the current token, when deciding

which production to apply [16, p. 146]. For the purposes of Rios we henceforth assume that lookahead will be equivalent to $k = 1$ for all parsing methods in consideration.

4.4.1. Ambiguity

Within grammars, ambiguity is one of the most important things to consider. In this context, it refers to a situation where a grammar can derive more than one parse tree for its terminal strings. The reason many compilers require an unambiguous grammar is because, ambiguous grammars make it so that the compiler cannot guarantee a unique translation for all inputs. [16, p. 121] An example of this can be seen below in Figure 4.3:

```
1  Expr → Expr  -  Expr
2         |  id
```

Figure 4.3.: An ambiguous grammar allowing for multiple parse trees. Example taken from Crafting a Compiler [16, p. 121].

This example shows an ambiguous grammar that allows for multiple different parse trees to be derived from the same input string. A simple example of this would be the string $id - id - id$. As is shown in Figure 4.4, the compiler with the above shown grammar, would have multiple ways of reaching the desired outcome.

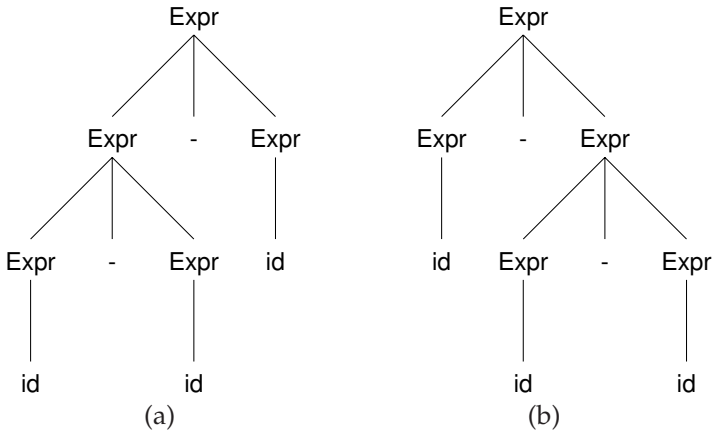

Figure 4.4.: Two different parse tree created from the same ambiguous grammar. Example taken from Crafting [16, p. 121].

Checking a CFG for ambiguity can be difficult, as there is no algorithm that can prove its unambiguity, for all CFG's. There are many tools that can check for ambiguity in

specific grammars, some of which were used in this project. These will be explained further in section 4.5. For this project, it was decided to make an LL(1) grammar, which cannot be ambiguous. Testing whether the grammar was LL(1) or not was done through the use of the tools COCO/R and PEG.js.

4.4.2. LR(1) Parser

An LR(1) parser employ the same base tactic of employing a simple set of methods that the parser continuously calls from. These actions help reduce expressions and facilitate the transfer of partial strings from two stacks. One approach to visualise the theory is to represent the process as two needles — the left needle which holds the processed part of the input string, and the right needle that has the unprocessed part of the string. The two commands employed in LR(1) parsers are:

- **Shift:** A method that transfers the next token from the right needle to the left, and records the state. [16, p. 182]
- **Reduce:** Replaces a token on the left needle with its corresponding left-hand-side symbol from the CFG. Formalised the reduction follows the rule $\mathbf{A} \rightarrow \gamma$. γ is a token that need to be converted to their corresponding left-hand-side symbol. **A** is the representation of the left-hand-side symbol. [16, p. 182]

4.4.3. LL(1) Parser

LL(1) parsers are a subset of LR(1) parsers, that restrict CFGs to make parsing easier. An LL(1) parser predicts the appropriate production for a non-terminal, by peeking at the *k* next terminal symbols (tokens) in the input stream. This is done via use of two commands:

- **Predict:** A method that predicts which production to apply for a given set of tokens, by utilising k lookahead. Formalised, the function reads as $Predict_k(p)$. *k* is the length of a set of token strings used to predict the string that comprises p. *p* is a grammar production expressed as a rule. It considers the production *p* and computes a set of strings with a length of exactly *k*, that will be used to predict the application of rule *p*. [16, p. 146]
- **Match:** A method that checks a stream of tokens for a specific token. Formalised, it is written as *match(ts,token)*, where *ts* is a stream of tokens, and *token* is the specific token we wish to check the stream against. *match()* peeks *k* tokens ahead, and if it finds a token corresponding to the specified *token*, it advances to the next token. [16, p. 149-150]

4.4.4. Comparing Parsing Techniques

Comparing LL(1) and LR(1) parsers can be difficult, as they both have strengths that are dependent on the context they are employed in. With regards to generality, it can be argued that LR(1) parsers are more powerful than LL(1) parsers, due to LR grammars being able to recognise a wider spectrum of grammars. This makes it more convenient because of the need to rewrite grammars occurring less frequently. Additionally, LR(k) has the added advantage of being able to handle left recursion.

On the other hand, when considering the ability to deal with error handling LL parsers shine. LL parsers make it easy to locate errors and recover from them. This is due to the way that every token is looked at in a meticulous and solitary fashion. Additionally, many programmers find it easy to create a LL(1) parser, or a parser with one token lookahead, by hand because of the restrictive nature of LL(1) grammars. [16, p. 171] However, one of the disadvantages of LL(1) parsers is that they have trouble handling left-recursion. Because the parser always picks the leftmost nonterminal, it will continue to evaluate the same nonterminal for perpetuity.

With regards to complexity, it is preferable to use an LL(1) parser. Recall the needles mentioned in section 4.4.2, and how tokens are thrown between stacks, to reverse productions to left-hand-side symbols to knit. This roundabout way of creating parse trees, by collecting them in chunks, where children nodes are knitted together, generally makes it harder to write in hand.

4.5. Parser Generating Tools

As mentioned in Appendix B there are multiple types of parsers, and correspondingly, there are many ways of creating a parser. Creating a parser by hand is a time-consuming process, and as such there are many tools that analyse and create parsers automatically. For Rios, multiple different tools were considered, all of which had pros and cons.

4.5.1. ANTLR

ANTLR is a free parsing generator used for creating LL(*) parsers. While both the tool and its documentation are free, the people behind ANTLR have created a book that shows how to utilise the parsing tool to its fullest. ANTLR is capable of creating three different types of recognisers: lexers, parsers and tree parsers. Creating a parser via ANTLR only requires that users provide a CFG of a given language. Said CFG is then used to create a lexer and parser, which in turn can be used to create a concrete syntax tree. [17] Older versions of ANTLR used to support the creation of abstract syntax trees (AST), but as of ANTLR4, creating an AST requires use of a listener or visitor pattern [18].

4.5.2. COCO/R

COCO/R is a tool meant for compiler generation. It takes a grammar for a given source language, whose productions are in extended backus-naur form and uses it to make a scanner and a parser. The produced scanner will work as a deterministic finite automaton, that can be configured to either account for case sensitivity or not. Parser construction will by default be LL(1) parser, but can also accept LL(K) since LL(1) conflicts can be resolved by using further lookahead. [19, p. 3-9]

4.5.3. Manual

A third option is to write a parser by hand. There are pros and cons associated with this, and they depend on the context and grammar being used.

One big advantage of writing a parser in hand is that it forces programmers to get an in-depth understand of the parsing tool being made. Because parsers are reliant on the employed CFG, it forces programmers to account for even the smallest changes in the grammar.

This in turn is also one of the disadvantages of writing a parser in hand. Creating a language is often an iterative process, which changes multiple times before the final language is created. Accordingly, this process often requires changes to the CFG, that the parser relies on. Due to this, a parser will likely have to be changed multiple times throughout a long-lasting projects.

For Rios the parser was made without the use of any automatic parser generating tools. This was due to there being some problems associated with both ANTLR and COCO/R. First, ANTLR would requires use of a form of dynamic LL CFG that would require the CFG to be changed. Secondly, COCO/R handles LL(1) conflicts by performing grammar transformations, that often reduce readability by changing various productions. This could be anything from changing enumerations to actual numbers, or reducing grammatical symbols to condensed pipe operator commands [19, p. 17-20].

4.6. The Rios Transformer

To make subsequent work with the parsing tree easier to handle, in the way of making the tree more uniform, a number of transformations are made on the tree. This is done by the transformer through a number of smaller tasks.

First, as mentioned in section 4.1.1, the transformer adds a global state around the programs entirety, to make code generation easier. Second, the transformer visits the various reactions and makes them uniform in their tree representation.

```
when A >=
    | 5 : //Do stuff
    | 7 : //Do other stuff

when (B == 5) : //Do stuff

always : //Do stuff
```

Figure 4.5.: Example reactions in Rios syntax

fig. 4.5 shows some example reactions in Rios. All are valid syntactically, but all of them could instead be expressed in the same form. The transformer takes care of this, changing the parser tree representations of the reactions to be uniform for ease of use in later stages.

```
when
    | A >= 5 : //Do stuff
    | A >= 7 : //Do other stuff

when | B == 5 : //Do stuff

when | true : //Do stuff
```

Figure 4.6.: Example reactions in Rios syntax

fig. 4.6 shows the transformed reactions, that are all still syntactically valid, in addition to being formatted in the same manner. The changes made simplify the reactions, both for semantic definition and for codegeneration. For example, the comparison operators have been moved into tokens. The shorthand version of the *when* reaction has been changed to resemble the longform version more. The *always* has been changed to a 'when' syntax that will simply always run.

4.7. Making a Symbol Table

Traditionally, a symbol table is a mechanism that allows the compiler to associate information with identifiers, and store them to be used across different modules. Every time a given identifier is declared and used, the accumulated information stored within the symbol table, will then be provided - regardless of where in which module is accessing it. Symbol table construction is done via a semantic-processing activity that traverses the AST to record every identifier and types found within the source program. [16, p. 18][16,

p. 48] Collection of preliminary information is traditionally done by a lexer module, but can also be delegated to other modules, as is the case with the Rios parser. The way Rios handles collection of information via multiple modules built around visitor patterns.

4.7.1. Utilising Visitor Patterns

Multiple phases of the Rios compiler modules utilise an implementation of a visitor design pattern. Visitor patterns are found within the transformer, symbol registrar, symbol checker and type checker modules. While implementation differs per module, the reasoning behind use remains the same. When using a compiler that represents programs as abstract syntax trees, like Rios does, it will need to perform operations on the various nodes of the AST. The compiler will have to employ different classes for the different node types. Assignment statements, arithmetic, or all require different node treatment, which is where visitor patterns prove useful. Visitor patterns pack all these operations into one object, allowing new operations to be added separately and making node classes independent of the operations that apply to them. The visitor pattern can then pass relevant operations to each node of the AST as it traverses the tree. This method allows for a system that is easier to understand, maintain and change, albeit operating a bit slower than traditional implementation. [20]

The transformer uses the visitor pattern to make transformations to the AST. Primarily, this concerns adding a global state to wrap the program in. Secondarily it transforms reactions in a few ways: condition preambles and case conditions are combined, while shorthand whens are extended into long forms. The symbol registrar uses the visitor pattern to register variables in the various scopes to the symbol table. The symbol checker uses the visitor pattern to ascertain that expressions using variables are using variables that already been declared. Finally, the type checker uses the visitor pattern to ascertain that expressions use valid types.

4.8. Reaction Sorting

As previously mentioned, Rios evaluates the most independent Reactions before dependant reactions. This is done with the use of Tarjan's strongly connected components algorithm [21]. The dependencies of reactions can be illustrated as a directed graph, such that reactions are vertices, and the dependencies are edges. Formally, given a directed graph $G = (V, E)$, an edge $(v, w) \in E$ means that reaction $v \in V$ writes to a variable which reaction $w \in V$ reads, meaning w is dependant on v. The use of Tarjan's strongly connected components algorithm results in a reverse topological sort of the reactions given, this is then converted into a topological sort. Allowing the reactions to be sorted by dependencies, with the most independent reaction at the head. In addition to this feature, the algorithm also detect cycles in the reaction dependencies, which would

otherwise defer from the given semantics of, changing a variable in multiple Reactions in a single state is not permitted. [21]

4.9. Code Generator

This section will go through the code generation module of the Rios compiler. It will explain how each state, reaction, and transition is handled at compile-time, in addition to how the individual code segments are transformed during compilation. This section will also explain each of these individually, as each concept can be translated independently of each of the other concepts. For example, how a reaction is translated has no impact on how a state is translated.

The code generator works by taking a Rios program and translating it to a Arduino Language program. This is done to make sure that Rios will be able to compile to any kind of Arduino unit, as this was one of the requirements for the language. This also makes it easier to implement a lot of the wanted features such as pins and serials, since it can translate the pins in Rios to pinmodes in Arduino Language. Similarly, it can translate Rios serials to Arduino Language serial connection declarations.

The code generation module of Rios works sequentially through various phases when it compiles a program. First, the `prepareFile` and `endFile` functions traverse through code supplied from previous modules. `prepareFile` creates function headers needed for the compilation of the program and, with regards to states, tries to find the lowest-level default child and all `OnEnter` calls in it and its parents. `endFile` works in continuation of `prepareFile` by supplying closing statements to the various processes created by `prepareFile`.

When the various groupings have been filled, the actual compilation process can begin. As Rios compiles down to Arduino Language, many of the custom constructs only found within Rios are converted into something very syntactically different, while remaining semantically equivalent.

4.9.1. Reactions

Rios handles reactions by converting them into if statements and Arduino Language functions. Each of the different Rios reactions, when, every, and always gets converted to if statements, while cases inside of the reactions are converted to functions that can be run when the condition is fulfilled

Listing 4.1: An example of a reaction in Rios.

```
1 //reaction:
2     |case condition : case statement//
```

Listing 4.2: An example of a converted reaction in Arduino Language

```
//if (condition){
    function call();
}
function name(){
    case statement;
}//
```

Listing 4.1 and Listing 4.2 shows how the code generator converts Rios code into Arduino Language code that the Arduino

4.9.2. States

The way Rios handles states is by converting them to switch cases that can be used within the Arduino Language. Every case within the switch statement corresponds to a state, that will be run when the `currentState` variable holds the value of the given case.

```
default state StateOne {
  // reaction //
  // reaction //
}

state StateTwo {
  // reaction //
  // reaction //
}
```

Listing 4.3: "A very simple example of states created within Rios."

```
enum states {
        NoState = -1,
        StateOne,
        StateTwo
};

states _currentState = NoState;
states _nextState = StateRising;

void loop() {
        switch(_currentState) {
                case StateOne:
                // Function //
                // Function //
                        break;
```

```
16                  case StateTwo:
17                  // Function //
18                  // Function //
19                          break;
20          }
21
22          if(_nextState != NoState) {
23                  _currentState = _nextState;
24                  _nextState = NoState;
25                  (*_tFunc)();
26          }
27  }
```

Listing 4.4: The Arduino Language equivalent of Listing 4.3.

The code shown in Listings 4.3 and 4.4 shows the conversion of states to a switch statement. State names are collected and inserted into an enum, and the contents of the enum are then used as the basis for the cases present in the switch statement used in Arduino. During compilation, Rios creates a point from which to begin, named `NoState` that contains no behavior. This is used as the starting point for the `currentState`, which the program then uses to springboard into the whichever states are prioritised at compile-time. The ordering of the cases found within the switch statement is based in the ordering of the states in the original Rios program, so that the state that is defined first in the Rios example will always correspond to the first case in the Arduino Language switch statement. The reactions within each case in the switch is placed in specific order, according to the reaction sorting as explained in section 4.8.

4.9.3. Transitions

Transitions between states in Rios is handled by transitions which corresponds to function pointers in C++.

```
1  void (*_tFunc)() = &_tFuncStartup;
2  // Other declarations, priority, setup, reaction loop.
3  void _tFuncStartup() {
4          StateGlobalOnenter();
5          StateOneOnenter();
6  }
```

To begin, the function points to a default function that runs the OnEnter functions needed to transition to the default state that the program enters. When the program begins, it skips the first run of the switch statement described in section 4.9.2, and encounters the transition check in the end.

The transition check executes a check on the value of `nextState`. If the value is not `NoState`, the transition is triggered. If the value is `NoState`, nothing is done. After this, the Arduino standard `loop` function restarts.

```
if(_nextState != NoState) {
        _currentState = _nextState;
        _nextState = NoState;
        (*_tFunc)();
}
```

In any given reaction, a transition may be triggered by setting `nextState` to any given state. When this is done, `tFunc` is also changed to a corresponding transition function.

```
if(_nextState == NoState || 0 > _priorities[_currentState][_nextState]) {
                        _nextState = StateOne;
                        _tFunc = &transitionGlobal2One;
// Other code
if(_nextState == NoState || 1 > _priorities[_currentState][_nextState]) {
                        _nextState = StateTwo;
                        _tFunc = &transitionGlobal2Two;
```

The transitions functions may in turn call a number of OnEnter functions. In this example, each only calls a single function.

```
void transitionGlobal2One() {
        StateRisingOnenter();
}
void transitionGlobal2Two() {
        StateFallingOnenter();
}
```

As has been commented on throughout the project there can, in some cases, exist two transitions to different states at the same time. The way the code generator handles this is via the `generatePriorities` and `visitTransitionStmtNode`. These functions create an array of $states * states$ size, where the array indexation matches the enum list mentioned in section 4.9.2. The first element in the enum would be `NoState` which would correspond to -1, with every following element corresponding to a value 1 higher than its predecessor. As an example, the corresponding mapping to a priority array with two states, would look like what is showcased in Listing 4.5 below.

```
int _priorities[3][3] = {
        {-1,0,1,},
        {-1,0,1,},
        {-1,0,1,},
};
```

Listing 4.5: Example priority array

These priorities are used when deciding whether or not to execute a state transition. First, the program checks whether the next state has been changed before. If it has, the priority of the new `nextState` is compared to the current `nextState`, with the state of higher priority being kept. Of note, is that a small compiler optimization is made at this point. As the conditions for a transition, and the priority for that transition is always known, a new `nextState` can be compared to a static value.

```
1  if(/*Condition*/) {
2          if(_nextState == NoState || 0 > _priorities[1][_nextState]) {
3                  _nextState = StateRising;
4                  _tFunc = &transitionGlobal2Rising;
5          }
6  }
7  else if(/*Other condition*/) {
8          if(_nextState == NoState || 1 > _priorities[1][_nextState]) {
9                  _nextState = StateFalling;
10                 _tFunc = &transitionGlobal2Falling;
11         }
12 }
```

Listing 4.6: Example priority check

4.10. Summary of Chapter 4

Found within this chapter was a thorough examination of compiler structure, and how relevant theory was applied to the Rios Compiler. Additionally, this chapter also gave readers and comprehensive understanding of why every module found within the Rios compiler, was designed to work as it does, and why the way of constructing it as such was utilising.

Readers should, after reading this chapter, have a firm understanding of how a compiler is structured; from when a source program is submitted to the compiler, until a new program comes out in a format fitting the target system. In addition to this, readers should understand that certain modules in the Rios compiler deviate from the norm described within section 4.1 due to the platform, and how the program structure is divided.

5. Evaluation

The goal of Rios was to create a language for micro-controllers which specialises in solving I/O-based problems. In order to guide the design of this, a number of requirements were presented. To find out whether we succeeded in creating such a language, it is necessary to test whether or not Rios is currently capable of using specialised I/O concepts to solve I/O problems, and how well it does this. Firstly, a practical test will be conducted, solely focusing on whether or not Rios is able to solve I/O problems. In the case that we are able to, an evaluation should be done analysing how well it does. For this purpose we re-use the language evaluation criteria with respect to I/O problems from section 2.3.

5.1. Practical Test

In section 2.2 the I/O automaton was presented, which included a number of concepts that, if used correctly, could solve I/O problems. These concepts were states, default states, actions and transitions, and served as the core mechanism to solve I/O-based problems. We now show that all these concepts are easily implemented using Rios.

```
1    state StateName {
2        // Code here
3    }
```

Figure 5.1.: Implementation of a state in Rios

```
1    default state StateName {
2        // Code here
3    }
```

Figure 5.2.: Implementation of a start state in Rios

```
    default state StateName {
        when x == 5 : // Code here
    }
```

Figure 5.3.: Implementation of a reaction in Rios

```
default state StateName {
    when x == 5 : enter(StateName2)
}

state StateName2 {
    // Declarations here
}
```

Figure 5.4.: Implementation of a transition in Rios

Given that each of the aforementioned concepts is implementable in Rios, it should be possible to simulate the behaviour of an I/O automaton, for any given I/O problem. We originally presented a smoke detector problem in section 2.2, the problem was solved using the Arduino language, the exact code to that solution being available in Appendix A. We now show that the same problem is solvable using Rios.

```
// Measure a value and change the behaviour when the measured value
    ↪ exceeds a given threshold. If the threshold is exceeded, a red LED
    ↪ should light up and a tone should play from a buzzer, otherwise a
    ↪ green LED should be lit and the buzzer should be silent.
dpin red = d@12
dpin green = d@11
apin buzzer = a@10
apin smoke = a@A5
serial sout = $usb 9600
int treshold = 400

always : [
  sout.println("Pin␣A0:␣");
  sout.println(smoke.read());
]

priority Alarm, Standby

default state Standby {
  when smoke.read() > treshold : transition(Alarm)
  onenter : [
    red.write(low);
    green.write(high);
    buzzer.write(0);
  ]
}

state Alarm {
  when smoke.read() < treshold : transition(Standby)
  onenter : [
    red.write(high);
    green.write(low);
    buzzer.write(100);
  ]
}
```

Figure 5.5.: An Rios implementation of the smoke-detector problem. (Arduino Language version shown in Appendix A)

5.2. Language Evaluation Criteria

The language evaluation criteria presented in section 2.3 were used to determine how well, the Arduino language and C++ were specialised for I/O based problems. After the creation of Rios, a language which goal was to be specialised for solving I/O problems, we apply the same evaluation criteria.

Simplicity: Given that Rios can implement all the relevant I/O specialised concepts in a simple and intuitive manner (see figures 5.1, 5.2, 5.3 and 5.4), the language naturally becomes simple to work with.

Orthogonality: Rios adds orthogonality mainly by targeting the communication source between the micro controller and the outside world. Comparable to the Arduino language's approach in having the pins locations declared as integers, and using these locations as parameters to various functions, Rios simply introduces new primitive data types for pins, such as apin and dpin. Furthermore, it is possible to construct complex data types such as sub states, by the utilisation of states.

Data type: Rios introduces a number of data types useful for solving I/O based problems, the most prominent ones being apin, dpin, and serial. These were added in order to support the necessary communication between the micro controller and its surroundings.

Syntax design: Keywords in Rios are designed as to hopefully let a given programmer gauge the semantic meaning of a keyword simply by reading it. For example, `"when"` and `"always"` are both keywords in Rios. However, `"when"` heavily implies a dependency, contrary to `"always"` which signals independence. Finally, adding multiple keywords to a variable should still let the programmer identify the properties of the variable. For instance, a normal state declaration is simply the word "state" followed by the state name. However, adding "default" before the "state" keyword, explicitly tells the user the state will be the initial state.

Abstraction: Rios allows for both process abstraction, through sub states and reactions, and data abstraction through states.

Expressively: Rios does have a level of expressively mainly in reactions, but also in operation calls. In order to simplify reading/writing reactions, it is possible to group multiple reactions using the same variables in their conditions. For example, rather than having,

```
1        when x == 5 : enter(State1)
2        when x == 3 : enter(State2)
```

Figure 5.6.: Two reactions dependent on the same variable written separately in Rios

The same can be achieved with,

```
1        when x ==
2        | 5: enter(State1)
3        | 3: enter(State2)
```

Figure 5.7.: Two reactions dependant on the same variable grouped in Rios

In 5.7 we group cases dependant on the same variable, allowing for better readability and writeability.

Type checking: With regards to type checking, this module handles it by checking whether a variable declaration matches its inferred type. This means that the type checker evaluates whether the values on both sides of equals signs match. This is done because of the lack of type coercion within Rios. Variables of the type int can therefore only be used with other int variables, and not double or float type variables. This same rule applies to the bytes and long types.

Exception handling: There is no availability for exception handling in the Rios language, since the language will run on microcontroller and there is no overhead on the device to support this feature.

Aliasing: The Rios language does not support aliasing. This is because there is no method to accessing the same piece of memory using two or more variable, since there is no support for pointer or similar methods.

5.3. Discussion

Rios was designed to specifically solve I/O based problems using I/O automata. This property was explored in section 5.1, where it was indicated that the usage of specialised I/O concepts in Rios, made it possible to solve I/O problems using I/O automata. It should however be noted, that while Rios was able to solve a simple I/O problem, it does not necessarily mean all I/O problems can be solved. However, given the illustration that Rios can easily implement all the specialised I/O concepts an I/O automaton requires, it is reasonable to presume that any problem which can be solved with an I/O automaton, can also be solved using Rios. For this reason, the first functional requirement has been satisfied. Similarly, given the simplicity and explicit in which the concepts of the I/O automaton are implemented, the first non-functional requirement has also been fulfilled. Furthermore, due to the nature of the compiler described in chapter 4, it is possible to compile Rios code to Arduino Language and therefore upload it to an Arduino micro-controller, meaning the second functional requirement is also complete.
In addition to this, given that Rios supports operations for both arithmetic and logical data types, the third functional requirement is also satisfied. Finally, because the design of Rios was guided by the need to solve I/O problems, more so than the need to run on a specific Arduino micro-controller, the resulting choices are transferable and servable

for any language which wishes to specialise in solving I/O based problems. Therefore, the third non-functional requirement is fulfilled. Lastly, due to the choice of utilising a recursive decent parser, it was possible to satisfy the final functional requirement, due to its property of being able to give the user detailed feedback on compilation error.

5.4. Conclusion

The purpose of this project was to create a specialised language for Arduino microcontrollers. The approach for this was to anchor the entire project around a problem statement:

> *Given that I/O automata are a common possible solution for I/O problems solved with Arduino microcontrollers, how can a language be designed which is based in the relevant aspects of the automata, and specialised for microcontrollers/Arduino?*

In order to accommodate this Rios was created, as an attempt to create a language that could handle I/O-based problems via I/O automata. Given that Rios was based in a specific problem solution to a limited problem domain, it naturally inherits the properties to solve said problems, as demonstrated in 5.5, and a further example can be seen in Appendix E. In order to evaluate the capacity with which it solves problems within the domain, certain criteria were applied in section 5.2 indicating that Rios is indeed specialised for I/O problems. Based on this, as well as the inherited properties of the I/O automata highlighted in earlier chapters, Rios was conclusively deemed an I/O-specialised language.

While Rios is a fully functioning language, it cannot be deemed complete. While Rios can compile many programs, certain functionalities are still to be desired. This could, for example, be the addition of statement control structures or proper testing suites. These ideas will be further explored in section 5.5.

In addition to the current implementation of Rios, the design choices applied to the language should in no way inhibit the possibility of the addition of other features. As the v1.0 implementation in no way interferes with any C++ constructions, Rios could in theory be possible to convert the language to be an extension to C++ rather than a separate language.

5.5. Future Work

While the creation of Rios was a thorough and meticulous process, it was still subject to severe time constraints due to being a student project. Because of this, some features

had to be implemented in a limited fashion, while others had to be cut entirely.

This section will explain some possibilities for further development should this project ever see an influx in time and resources for such a course.

5.5.1. Extensive Testing Suites

The amount of testing done to secure correct implementation in Rios was less than sufficient. On launch of version 1.0, the amount of tests included was limited to *JUnit5* unit-tests that verified whether the output of individual strings was as intended. Because v1.0 only tests the individual lines, there is no assurance that entire modules would work according to the collective semantic meaning. Because of this, Rios would benefit greatly from implementing extensive testing suites to deal with integration tests.

Additionally, v1.0 only represents the uses intended by the creators of Rios. All testing was done in an isolated environment, where intended uses were known, and semantic meaning was implicitly understood by users. Because of this, it would be beneficial to commit resources in a future version to do some semblance of implementation testing. Testing the language on users not familiar with it would potentially help find mistakes or semantic inconsistencies not considered during v1.0 development.

Translation Validation

There are many ways of making sure that a compiler works as intended. It is, however, still not possible to automatically prove that a compiler always produces the desired result, that is equivalent to the source program [22]. This is because of the fact that a near-infinite amount of programs that can be compiled. There are however ways of making sure that each compilation achieves the desired result. One such way is translation validation. The idea behind translation validation is to check the compiled program against the source program, and pinpoint potential differences. Translation validation serves as a tool to help raise effectiveness and automate the testing process. While it does not remove the need for other testing suites, where the actual testing takes place, it does remove the explicit need of looking through the compiled results, to pinpoint errors. It does this by analysing the program steps via a two-step inference algorithm that utilises control-flow graphs, as well as a symbolic evaluation.[22]

As was explained, in section 5.5.1 the amount of tests found within Rios was somewhat basic. If given more time, it would prove very beneficial to improve further upon the increased amount of test suites, by adding verification to the various suites being used to ensure correct compilation. Translation validation, while still a prototype, could be one potential candidate that Rios would benefit greatly from implementing.

5.5.2. User-functions

One of the things that were originally planned for the Rios was to include user-defined methods. While users can use implemented syntax like `when`s to manipulate data, there is no way for a user to properly create a custom method to use within a program. While v1.0 of Rios can handle I/O problems, the language cannot truly be considered 'done' without this feature. This means that if a user has to employ the same method twice, that user will have to write that method within the program twice, as there is no way to call the same method multiple times.

5.5.3. Control Structures

Another limitation of the v1.0 of Rios was the lack of control structures in the traditional sense. While it supports the use of `when`s it lacks any and all support of `if-else` statements, as well as `while` and `for` loops in statements. While v1.0 can be said to work, the lack of statement control structures, limits the range of actions a programmer can commit to when creating a program.

5.5.4. Type Coercion

The v1.0 of Rios has no type coercion, as is also reflected in the type rules in Appendix D. However, as there are several types that can be used for integers, one might expect that these could be used together. For example, adding a byte value onto a variable of type long should internally convert the byte to a long and add them together. Similarly, as pin readings can only return ints and bools, a programmer wanting holding variables for the pin values is forced to use variables of those types. Allowing simple type coercion would fix many of these problems, while also being relatively simple to implement.

Bibliography

[1] Steven F. Barrett.
Arduino Microcontroller Processing for Everyone! Third Edition.
Morgan & Claypool, 2013,
Pp. 9–16.
ISBN: 9781627052535.

[2] The Wikipedia Community.
List of common microcontrollers.
URL: https://en.wikipedia.org/wiki/List_of_common_microcontrollers.
Last retrieved 21-02-2019.

[3] Julia Lobur Linda Null.
The Essentials of Computer Organization and Architecture.
Jones and Bartlett Learning, 2013,
Chapter 7.3.
ISBN: 9780763704445.

[4] Arduino.
Introduction.
URL: https://www.arduino.cc/en/Guide/Introduction.
Last retrieved 14-02-2019.

[5] Sparkfun.
What is an Arduino?
URL: https://learn.sparkfun.com/tutorials/what-is-an-arduino/all#whats-on-the-board.
Last retrieved 14-02-2019.

[6] Arduino Organization.
Arduino Project Hub.
URL: https://create.arduino.cc/projecthub.
Last retrieved 06-05-2019.

[7] Arduino.
What is Arduino?
URL: https://www.arduino.cc/en/Guide/Introduction.
Last retrieved 27-03-2019.

[8] Alfred W. England.
Computer input-output system.
URL: https://patents.google.com/patent/US3702462A/en.

Last retrieved 08-04-2019.

[9] Nancy Lynch and Mark Tuttle.
An Introduction to Input/Output automata.
Technical Memo MIT/LCS/TM-373.
Massachusetts Institute of Technology, Nov. 1988.

[10] Robert W. Sebesta.
Concepts of Programming Languages - Eleventh Edition.
Pearson Education Limited, 2016,
Pp. 30–41.
ISBN: 978-1-292-10055-5.

[11] Arduino.
Libraries.
URL: https://www.arduino.cc/en/reference/libraries.
Last retrieved 29-03-2019.

[12] John Foxworth.
HOW TO PROGRAM A MICROCONTROLLER.
URL: https://www.egr.msu.edu/classes/ece480/capstone/spring15/group13/assets/app_note_john_foxworth.docx.pdf.
Last retrieved 28-03-2019.

[13] Caleb Helbling and Samuel Z. Guyer.
Juniper: A Functional Reactive Programming Language for the Arduino.
URL: http://www.juniper-lang.org/publications/juniper_farm_preprint.pdf.
Last retrieved 09-04-2019.

[14] Juniper.
Tutorial.
URL: http://www.juniper-lang.org/tutorial.html.
Last retrieved 04-05-2019.

[15] David Benyon.
Designing Interactive Systems.
Pearson Education Limited, 2014,
P. 50.
ISBN: 978-1-4479-2011-3.

[16] Richard J. LeBlanc Jr. Charles N. Fischer Ron K. Cytron.
Crafting a Compiler.
Pearson Education Limited, 2010,
P. 683.
ISBN: 978-0-13-606705-4.

[17] Ben Hamilton Terrence Parr.
ANTLR Documentation.
URL: https://github.com/antlr/antlr4/blob/master/doc/faq/general.md.

Last retrieved 06-05-2019.

[18] Terrence Parr.
ANTLR Listener Documentation.
URL: https://github.com/antlr/antlr4/blob/master/doc/faq/parse-trees.md#what-if-i-need-asts-not-parse-trees-for-a-compiler-for-example.
Last retrieved 06-05-2019.

[19] Hanspeter Mössenböck.
The Compiler Generator Coco/R: User Manual.
URL: http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf.
Last retrieved 06-05-2019.

[20] John Vlissides Erich Gamma Richard Helm Ralp Johnson.
Design Patterns.
Pearson Education, 1995,
Pp. 331–332.
ISBN: 978-0-201-63361-0.

[21] Robert Tarjan.
"Depth first search and linear graph algorithms".
In: SIAM JOURNAL ON COMPUTING 1.2 (1972).

[22] George C. Necula.
Translation Validation for an Optimizing Compiler.
URL: https://people.eecs.berkeley.edu/~necula/Papers/tv_pldi00.pdf.
Last retrieved 13-05-2019.

Appendices

A. Smoke/Gas Detector Code

```
/*******

 All the resources for this project:
 https://www.hackster.io/Aritro

*******/

int redLed = 12;
int greenLed = 11;
int buzzer = 10;
int smokeA0 = A5;
// Your threshold value
int sensorThres = 400;

void setup() {
  pinMode(redLed, OUTPUT);
  pinMode(greenLed, OUTPUT);
  pinMode(buzzer, OUTPUT);
  pinMode(smokeA0, INPUT);
  Serial.begin(9600);
}

void loop() {
  int analogSensor = analogRead(smokeA0);

  Serial.print("Pin A0: ");
  Serial.println(analogSensor);
  // Checks if it has reached the threshold value
  if (analogSensor > sensorThres)
  {
    digitalWrite(redLed, HIGH);
    digitalWrite(greenLed, LOW);
    tone(buzzer, 1000, 200);
  }
  else
  {
```

```
37      digitalWrite(redLed, LOW);
38      digitalWrite(greenLed, HIGH);
39      noTone(buzzer);
40    }
41    delay(100);
42  }
```

B. Context Free Grammar

$$
\begin{aligned}
S &::= Decs\ EOF \\
Decs &::= \{Dec\} \\
Dec &::= VarDec \\
&\mid ReactDec \\
&\mid StateDec \\
&\mid OnEnter \\
&\mid PriorityList \\
StateDec &::= [\texttt{default}]\ \texttt{state}\ StateId\ \texttt{\{}\ Decs\ \texttt{\}} \\
VarDec &::= Type\ Id\ Assign\ Expr \\
ReactDec &::= When \\
&\mid Always \\
&\mid Every \\
PriorityList &::= \texttt{priority}\ StateId\ \{\texttt{,}\ StateId\} \\
Every &::= \texttt{every}\ IntLiteral\ Unit\ (Result \mid When) \\
Always &::= \texttt{always}\ Result \\
OnEnter &::= \texttt{onenter}\ Result \\
When &::= \texttt{when}\ ExprNum\ (Result \mid [(CompOp|BoolOp)]\ Cases) \\
Cases &::= Case\ \{Case\} \\
Case &::= \texttt{|}\ [(CompOp|BoolOp)]\ Expr\ Result \\
Result &::= \texttt{:}\ Stmts
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} &::= \mathit{ExprBool} \\
\mathit{ExprBool} &::= \mathit{ExprComp}\ [\mathit{BoolOp}\ \mathit{ExprBool}] \\
\mathit{ExprComp} &::= \mathit{ExprNum}\ [\mathit{CompOp}\ \mathit{ExprComp}] \\
\mathit{ExprNum} &::= \mathit{ExprProduct}\ [\mathit{NumOp1}\ \mathit{ExprNum}] \\
\mathit{ExprProduct} &::= \mathit{ExprNegate}\ [\mathit{NumOp2}\ \mathit{ExprProduct}] \\
\mathit{ExprNegate} &::= [\texttt{!}]\ \mathit{ExprCall} \\
\mathit{ExprCall} &::= \mathit{ExprSingle}\ [[\texttt{.}\ \mathit{Calls}]\ \texttt{(}\ [\mathit{Expr}\ \{\texttt{,}\ \mathit{Expr}\}]\ \texttt{)}] \\
\mathit{ExprSingle} &::= \mathit{Literal} \\
&\quad | \ \mathit{Id} \\
&\quad | \ \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
\mathit{Literal} &::= \mathit{IntLiteral} \\
&\quad | \ \mathit{ByteLiteral} \\
&\quad | \ \mathit{LongLiteral} \\
&\quad | \ \mathit{FloatLiteral} \\
&\quad | \ \mathit{PinLiteral} \\
&\quad | \ \mathit{SerialLiteral} \\
&\quad | \ \mathit{StringLiteral} \\
&\quad | \ \mathit{BoolLiteral}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Stmts} &::= \texttt{[}\ \mathit{Stmt}\ \{\texttt{;}\ [\mathit{Stmt}]\}\ \texttt{]} \\
&\quad | \ \mathit{Stmt} \\
\mathit{Stmt} &::= \mathit{VarDec} \\
&\quad | \ \mathit{TransitionStmt} \\
&\quad | \ \mathit{Expr}\ [\{\texttt{,}\ \mathit{Expr}\}\ (\mathit{CompAssign}|\mathit{Assign})\ \mathit{Expr}] \\
\mathit{TransitionStmt} &::= \texttt{transition}\ \texttt{(}\ \mathit{StateId}\ \texttt{)} \\
\mathit{Calls} &::= \texttt{read} \\
&\quad | \ \texttt{toggle} \\
&\quad | \ \texttt{write} \\
&\quad | \ \texttt{println}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{BoolLiteral} &::= \texttt{true} \mid \texttt{false} \mid \texttt{high} \mid \texttt{low} \\
\mathit{PinLiteral} &::= (\texttt{a} \mid \texttt{analog} \mid \texttt{d} \mid \texttt{digital})\ \texttt{@}\ [\texttt{A}]\ \mathit{IntLiteral}\ [\texttt{pullup}] \\
\mathit{SerialLiteral} &::= \texttt{\$}\ (\texttt{usb} \mid \texttt{pin})\ \mathit{IntLiteral} \\
\mathit{StringLiteral} &::= \texttt{"}\ \{\mathit{notQuote}\}\ \texttt{"} \\
\mathit{IntLiteral} &::= \mathit{digit}\ \{\mathit{digit}\} \\
\mathit{LongLiteral} &::= \mathit{digit}\ \{\mathit{digit}\}\ \texttt{L} \\
\mathit{ByteLiteral} &::= \mathit{digit}\ \{\mathit{digit}\}\ \texttt{B} \\
\mathit{FloatLiteral} &::= \mathit{digit}\ \{\mathit{digit}\}\ \texttt{.}\ [\mathit{digit}\ \{\mathit{digit}\}]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{CompOp} &::= \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} \\
\mathit{BoolOp} &::= \texttt{\&\&} \mid \texttt{||} \\
\mathit{NumOp1} &::= \texttt{+} \mid \texttt{-} \\
\mathit{NumOp2} &::= \texttt{*} \mid \texttt{/} \\
\mathit{Assign} &::= \texttt{=} \\
\mathit{CompAssign} &::= (\mathit{NumOp1} | \mathit{NumOp2})\ \mathit{Assign} \\
\mathit{Id} &::= \mathit{smallLetter}\ \{\mathit{anyFollowing}\} \\
\mathit{StateId} &::= \mathit{bigLetter}\ \{\mathit{anyFollowing}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Unit} &::= \texttt{d} \mid \texttt{h} \mid \texttt{m} \mid \texttt{s} \mid \texttt{ds} \mid \texttt{cs} \mid \texttt{ms} \\
\mathit{Type} &::= \texttt{apin} \mid \texttt{dpin} \mid \texttt{serial} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{byte} \mid \texttt{float} \mid \texttt{long} \\
\mathit{notQuote} &::= \mathit{ANY}\ -\ \texttt{"}\ -\ \mathit{eol} \\
\mathit{digit} &::= \texttt{0} \ldots \texttt{9} \\
\mathit{eol} &::= \texttt{\textbackslash n} \\
\mathit{smallLetter} &::= \texttt{a} \ldots \texttt{z} \\
\mathit{bigLetter} &::= \texttt{A} \ldots \texttt{Z} \\
\mathit{anyFollowing} &::= \texttt{a} \ldots \texttt{z} + \texttt{A} \ldots \texttt{Z}
\end{aligned}
$$

C. Structural operational semantics

Abstract syntax categories:

n	$\in$ **Num**	Numerals
x	$\in$ $\mathbf{Name_V}$	Variable Names
s	$\in$ $\mathbf{Name_S}$	State Names
Dp	$\in$ $\mathbf{Dec_P}$	Priority List Declarations
Do	$\in$ $\mathbf{Dec_O}$	OnEnter Declarations
Dv	$\in$ $\mathbf{Dec_V}$	Variable Declarations
Ds	$\in$ $\mathbf{Dec_S}$	State Declarations
Dr	$\in$ $\mathbf{Dec_R}$	Reaction Declarations
Dc	$\in$ $\mathbf{Dec_C}$	Reaction Case Declarations
S	$\in$ **Stmt**	Statements
e	$\in$ **Expr**	Expressions
u	$\in$ **Unit**	Time units
T	$\in$ **Type**	Types
p	$\in$ **Prog**	Program

$$
\begin{aligned}
e &::= n \mid x \mid \texttt{(}\ e\ \texttt{)} \mid e_1\ \texttt{+}\ e_2 \mid e_1\ \texttt{-}\ e_2 \mid e_1\ \texttt{*}\ e_2 \mid e_1\ \texttt{/}\ e_2 \\
&\quad \mid e_1\ \texttt{==}\ e_2 \mid e_1\ \texttt{!=}\ e_2 \mid e_1\ \texttt{<}\ e_2 \mid e_1\ \texttt{>}\ e_2 \mid e_1\ \texttt{<=}\ e_2 \mid e_1\ \texttt{>=}\ e_2 \\
&\quad \mid \texttt{!}e \mid e_1\ \texttt{||}\ e_2 \mid e_1\ \texttt{\&\&}\ e_2 \\
Dp &::= s\texttt{,}\ Dp \mid s \mid \varepsilon \\
Do &::= \texttt{onenter:}\ S \mid \varepsilon \\
Dv &::= t\ x\ \texttt{=}\ e\ Dv \mid \varepsilon \\
Ds &::= \texttt{state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds\ \texttt{\}}\ Ds \mid \varepsilon \\
Dr &::= \texttt{when}\ c \mid \varepsilon \\
Dc &::= \texttt{|}\ e\ \texttt{:}\ S\ c \mid \varepsilon \\
T &::= \texttt{apin} \mid \texttt{dpin} \mid \texttt{serial} \mid \texttt{int} \mid \texttt{long} \mid \texttt{bool} \mid \texttt{byte} \mid \texttt{float} \\
S &::= S_1\ \texttt{;}\ S_2 \mid x\ \texttt{=}\ e \mid Dv \mid \texttt{transition(}\ s_1\ \texttt{)} \mid e \mid \varepsilon \\
p &::= Ds
\end{aligned}
$$

C.1. Meta-sets

(C.1) $$v \in \mathbf{Val} = \mathbb{Q} \cup Strings \cup \{true, false\}$$

(C.2) $$r \in \mathbf{Rea} = (\mathbf{Dec_C} \times \mathbf{Var} \times \mathbf{Pri})$$

(C.3) $$r^* \in \mathbf{Rea}^*$$

(C.4) $$next_s \in \mathbf{Name_S} \times \{\varepsilon\}$$

(C.5) $$curr_s \in \mathbf{Name_S}$$

(C.6) $$l \in \mathbf{Loc}$$

(C.7) $$\mathbf{Name_{S\mathcal{E}}} = \mathbf{Name_S} \cup \{\varepsilon\}$$

C.2. Helping Functions

(C.8) $$\mathcal{N} : \mathbf{Num} \to \mathbb{Q}$$

(C.9) $$pri \in \mathbf{Pri} : (\mathbf{Name_S} \cup \{next\}) \rightharpoonup \mathbb{N}$$

(C.10) $$var \in \mathbf{Var} : \mathbf{Name_V} \rightharpoonup \mathbf{Loc}$$

(C.11) $$sto \in \mathbf{Sto} : \mathbf{Loc} \rightharpoonup \mathbf{Val}$$

(C.12) $$sta \in \mathbf{Sta} : \mathbf{Name_S} \rightharpoonup \underbrace{\mathbf{Rea}^*}_{Reactions} \times \underbrace{\mathbf{Stmt}}_{OnEnter} \times \underbrace{\mathbf{Name_{S\mathcal{E}}}}_{Parent} \times \underbrace{\mathbf{Name_{S\mathcal{E}}}}_{DefaultChild}$$

(C.13) $$anc(s) = \{s\} \cup anc(pare_s) \textbf{ where } (pare_s, ...) = sta(s)$$

(C.14) $$rel \in \mathbf{Rel} : (\mathbf{Rea} \times \{reader,\ writer\}) \rightharpoonup \mathbf{Loc}^*$$

(C.15) $$nextsto(l) = l + 1$$

C.3. Transition Systems

$$
\begin{aligned}
&\textit{declaration transitions system} = \\
&\qquad \rightarrow_{Dp} = \underbrace{(\mathbf{Dec_P} \times \mathbf{Pri})}_{\Gamma_{Dp}} \times \underbrace{(\mathbf{Pri})}_{T_{Dp}} \\
&\qquad\qquad \textit{var}, \textit{rel}, \textit{pri}, \textit{pare}_s \vdash \langle \mathbf{Dec_P}, \textit{sto}, \textit{sta} \rangle \rightarrow_{Dp} \langle \textit{sto}, \textit{sta} \rangle \\
&\qquad \rightarrow_{Do} = (\mathbf{Dec_O} \times \mathbf{Sta}) \times (\mathbf{Sta}) \\
&\qquad\qquad s \vdash \langle \mathbf{Dec_O}, \textit{sta} \rangle \rightarrow_{Do} \langle \textit{sta} \rangle \\
&\qquad \rightarrow_{Dv} = (\mathbf{Dec_V} \times \mathbf{Var} \times \mathbf{Sto}) \times (\mathbf{Var} \times \mathbf{Sto}) \\
&\qquad\qquad \langle \mathbf{Stmt}, \textit{pri} \rangle \rightarrow_{Dv} \langle \textit{pri} \rangle \\
&\qquad \rightarrow_{Dr} = (\mathbf{Dec_R} \times \mathbf{Sta} \times \mathbf{Rel}) \times (\mathbf{Sta} \times \mathbf{Rel}) \\
&\qquad\qquad \textit{var},\ \textit{sto}, \textit{pri}, s \vdash \langle \mathbf{Dec_R}, \textit{sta}, \textit{rel} \rangle \rightarrow_{Dr} \langle \textit{sta}, \textit{rel} \rangle \\
&\qquad \rightarrow_{Ds} = (\mathbf{Dec_S} \times \mathbf{Sto} \times \mathbf{Sta}) \times (\mathbf{Sto} \times \mathbf{Sta}) \\
&\qquad\qquad \langle \textit{sto}, \textit{sta} \rangle \rightarrow_{Ds} \langle \textit{sto}, \textit{sta} \rangle \\
&\qquad \rightarrow_{rel} = (\mathbf{Stmt} \times \mathbf{Var} \times \mathbf{Rel}) \times (\mathbf{Var} \times \mathbf{Rel}) \\
&\qquad\qquad \langle \textit{var}, \textit{rel} \rangle \rightarrow_{rel} \langle \textit{var}, \textit{rel} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\textit{global transitions system} = (\Gamma_g,\ \Rightarrow_g) \\
&\qquad \Gamma_g = \mathbf{Dec_S} \cup (\mathbf{Rea}^* \times \mathbf{Sto} \times \underbrace{\mathbf{Name_S}}_{curr_s} \times \underbrace{\mathbf{Name_{S\mathcal{E}}}}_{next_s}) \\
&\qquad \Rightarrow_g = \Gamma_g \times \Gamma_g
\end{aligned}
$$

$$
\begin{aligned}
&\textit{transition transition system} = (\Gamma_t,\ T_t,\ \rightarrow_t) \\
&\qquad \Gamma_t = \mathbf{Name_S} \times \mathbf{Sto} \times \mathbf{Name_{S\mathcal{E}}} \\
&\qquad T_t = \mathbf{Sto} \times \mathbf{Name_{S\mathcal{E}}} \\
&\qquad \rightarrow_t = \Gamma_t \times T_t \\
&\qquad\qquad curr_s, next_s \vdash \langle s, \textit{var}, \textit{sto}, next_s \rangle \rightarrow_t \langle \textit{var}, \textit{sto}, next_s \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\textit{local transition system} = (\Gamma_l,\ T_l,\ \rightarrow_l) \\
&\qquad \Gamma_l = \mathbf{Stmt} \times \mathbf{Var} \times \mathbf{Sto} \times \mathbf{Name_{S\mathcal{E}}} \\
&\qquad T_l = \mathbf{Var} \times \mathbf{Sto} \times \mathbf{Name_{S\mathcal{E}}} \\
&\qquad \rightarrow_l = \Gamma_l \times T_l
\end{aligned}
$$

$$
\begin{aligned}
\textit{expression transition system} &= (\Gamma_e,\ T_e,\ \ \rightarrow_e) \\
\Gamma_e &= \mathbf{Expr} \\
T_e &= \mathbf{Val} \\
\rightarrow_e &= \Gamma_e \times T_e
\end{aligned}
$$

C.4. Declaration

$$
\begin{aligned}
&\textit{declaration transitions system} = \\
&\qquad \rightarrow_{Dp} = \underbrace{(\mathbf{Dec_P} \times \mathbf{Pri})}_{\Gamma_{Dp}} \times \underbrace{(\mathbf{Pri})}_{T_{Dp}} \\
&\qquad\qquad var, rel, pri, pare_s \vdash \langle \mathbf{Dec_P}, sto, sta \rangle \rightarrow_{Dp} \langle sto, sta \rangle \\
&\qquad \rightarrow_{Do} = (\mathbf{Dec_O} \times \mathbf{Sta}) \times (\mathbf{Sta}) \\
&\qquad\qquad s \vdash \langle \mathbf{Dec_O}, sta \rangle \rightarrow_{Do} \langle sta \rangle \\
&\qquad \rightarrow_{Dv} = (\mathbf{Dec_V} \times \mathbf{Var} \times \mathbf{Sto}) \times (\mathbf{Var} \times \mathbf{Sto}) \\
&\qquad\qquad \langle \mathbf{Stmt}, pri \rangle \rightarrow_{Dv} \langle pri \rangle \\
&\qquad \rightarrow_{Dr} = (\mathbf{Dec_R} \times \mathbf{Sta} \times \mathbf{Rel}) \times (\mathbf{Sta} \times \mathbf{Rel}) \\
&\qquad\qquad var,\ \ sto, pri, s \vdash \langle \mathbf{Dec_R}, sta, rel \rangle \rightarrow_{Dr} \langle sta, rel \rangle \\
&\qquad \rightarrow_{Ds} = (\mathbf{Dec_S} \times \mathbf{Sto} \times \mathbf{Sta}) \times (\mathbf{Sto} \times \mathbf{Sta}) \\
&\qquad\qquad \langle sto, sta \rangle \rightarrow_{Ds} \langle sto, sta \rangle \\
&\qquad \rightarrow_{rel} = (\mathbf{Stmt} \times \mathbf{Var} \times \mathbf{Rel}) \times (\mathbf{Var} \times \mathbf{Rel}) \\
&\qquad\qquad \langle var, rel \rangle \rightarrow_{rel} \langle var, rel \rangle
\end{aligned}
$$

(DEC-STATE-DEFAULT)

$$\frac{var, sto, pri, pare_s \vdash \langle \texttt{state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds_1\ \texttt{\}}\ Ds_2,\ sto,\ sta' \rangle \rightarrow_{Ds} \langle sto', sta'' \rangle}{var, sto, pri, pare_s \vdash \langle \texttt{default state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds_1\ \texttt{\}}\ Ds_2,\ sto,\ sta \rangle \rightarrow_{Ds} \langle sto', sta'' \rangle}$$

where $(p_r, p_o, p_p, p_c) = sta(pare_s)$ **and** $p_c = \varepsilon$ **and** $sta' = sta[pare_s \mapsto (p_r, p_o, p_p, s)]$

(DEC-STATE)

$$\frac{\begin{array}{r} var, rel, pri, pare_s \vdash \langle Dp,\ pri \rangle \rightarrow_{Dp} \langle pri' \rangle \\ s \vdash \langle Do,\ sta' \rangle \rightarrow_{Do} \langle sta'' \rangle \\ \langle Dv,\ var,\ sto \rangle \rightarrow_{Dv} \langle var',\ sto' \rangle \\ var', sto', pri', s \vdash \langle Dr,\ sta'',\ rel \rangle \rightarrow_{Dr} \langle sta''',\ rel' \rangle \\ var', sto', pri', s \vdash \langle Ds_1,\ sto',\ sta''' \rangle \rightarrow_{Ds} \langle sto'',\ sta'''' \rangle \\ var, sto, pri, pare_s \vdash \langle Ds_2,\ sto'',\ sta'''' \rangle \rightarrow_{Ds} \langle sto''',\ sta''''' \rangle \end{array}}{var, sto, pri, pare_s \vdash \langle \texttt{state}\ s\ \texttt{\{}\ Dp\ Do\ Dv\ Dr\ Ds_1\ \texttt{\}}\ Ds_2,\ sto,\ sta \rangle \rightarrow_{Ds} \langle sto''', sta''''' \rangle}$$

where $(p_r, p_o, p_p, p_c) = sta(pare_s)$ **and** $sta' = sta[s \mapsto (p_r, \varepsilon, pare_s, \varepsilon)]$

(DEC-STATE-EMPTY) $\quad var, sto, pri, pare_s \vdash \langle \varepsilon,\ sto,\ sta \rangle \rightarrow_{Ds} \langle sto,\ sta \rangle$

(DEC-PRIORITY-LIST)

$$\frac{var, rel, pri, pare_s \vdash \langle Dp,\ pri \rangle \rightarrow_{Dp} \langle pri' \rangle \quad \langle s,\ pri' \rangle \rightarrow_{Dp} \langle pri'' \rangle}{var, rel, pri, pare_s \vdash \langle s\texttt{,}\ Dp,\ pri \rangle \rightarrow_{Dp} \langle pri'' \rangle}$$

(DEC-PRIORITY-ELEMENT)

$var, rel, pri, pare_s \vdash \langle s,\ pri \rangle \rightarrow_{Dp} \langle pri[s \mapsto p][next \mapsto p + 1] \rangle$ **where** $p = pri(next)$

(DEC-ONENTER)

$$\frac{s \vdash \langle \texttt{onenter :}\ S, sta \rangle \rightarrow_{Do} \langle sta' \rangle}{\textbf{where}\ sta' = sta[s \mapsto (r^*, S, pare_s, child_s)]}\ \textbf{and}\ (r^*, \varepsilon, pare_s, child_s) = sta(s)$$

(DEC-ONENTER-EMPTY) $\quad s \vdash \langle \varepsilon, sta \rangle \rightarrow_{Do} \langle sta \rangle$

(DEC-VAR)

$$\frac{var, sto \vdash \langle e,\ var,\ sto \rangle \rightarrow_e v \quad \langle Dv,\ var,\ sto \rangle \rightarrow_{Dv} \langle var',\ sto' \rangle}{\langle t\ x\ \texttt{=}\ e\ Dv,\ var,\ sto \rangle \rightarrow_{Dv} \langle var'[x \mapsto l],\ sto'[l \mapsto v][next \mapsto next(l)] \rangle}\ \textbf{where}\ l = var[next]$$

(DEC-VAR-EMPTY) $\quad \langle \varepsilon,\ var,\ sto \rangle \rightarrow_{Dv} \langle var,\ sto \rangle$

(DEC-REACTION)

$$\frac{\langle c,\ rel\rangle \rightarrow_{rel} \langle rel'\rangle \quad var, sto, pri, s \vdash \langle Dr,\ sta',\ rel'\rangle \rightarrow_{Dr} \langle sta'',\ rel''\rangle}{var, sto, pri, s \vdash \langle \texttt{when}\ c\ Dr,\ sta,\ rel\rangle \rightarrow_{Dr} \langle sta'', rel''\rangle}$$

$$\textbf{where}\ sta' = sta[s \mapsto sta(s) \cup \{(c,\ var)\}]$$

(DEC-REACTION-EMPTY) $\quad var, sto, pri, s \vdash \langle \varepsilon, sta, rel\rangle \rightarrow_{Dr} \langle sta, rel\rangle$

(DEC-CAS) $\quad \dfrac{\langle S, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle \quad \langle c, var, rel'\rangle \rightarrow_{rel} \langle var, rel''\rangle}{\langle e\ \texttt{:}\ S\ c, var, rel\rangle \rightarrow_{rel} \langle var, rel''\rangle}$

(DEC-CAS-EMPTY) $\quad \langle \varepsilon, var, rel\rangle \rightarrow_{rel} \langle var, rel\rangle$

(DEC-STMT-IGNORE)

$$\langle S, var, rel\rangle \rightarrow_{rel} \langle var, rel\rangle \quad \textbf{where}\ S \in \{\varepsilon,\ \texttt{transition(}\ s\ \texttt{)}\}$$

(DEC-STMT-VAR) $\quad \dfrac{\langle e, var, rel\rangle \rightarrow_{rel} \langle var', rel'\rangle}{\langle t\ x\ \texttt{=}\ e, var, rel\rangle \rightarrow_{rel} \langle var'[x \mapsto var(next)], rel'\rangle}$

(DEC-STMT-ASS)

$$\frac{\langle e, var, rel\rangle \rightarrow_{rel} \langle var', rel'\rangle}{\langle x\ \texttt{=}\ e, var, rel\rangle \rightarrow_{rel} \langle var', rel'[(r, writer) \mapsto rel(r, writer) \cup \{l\}]\rangle} \quad \textbf{where}\ l = var(x)$$

(DEC-STMT-SEQ) $\quad \dfrac{\langle S_1, var, rel\rangle \rightarrow_{rel} \langle var', rel'\rangle \quad \langle S_2, var', rel'\rangle \rightarrow_{rel} \langle var'', rel''\rangle}{\langle S_1\ \texttt{;}\ S_2, var, rel\rangle \rightarrow_{rel} \langle var'', rel''\rangle}$

(DEC-EXPR-BIN) $\quad \dfrac{\langle e_1, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle \quad \langle e_1, var, rel'\rangle \rightarrow_{rel} \langle var, rel''\rangle}{\langle e_1\ op\ e_2, var, rel\rangle \rightarrow_{rel} \langle var, rel''\rangle}$

$$\textbf{where}\ op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{||}, \texttt{\&\&}\}$$

(DEC-EXPR-SIN) $\quad \dfrac{\langle e, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle}{\langle \texttt{!}e, var, rel\rangle \rightarrow_{rel} \langle var, rel'\rangle}$

(DEC-EXPR-NUM) $\quad \langle n, var, rel\rangle \rightarrow_{rel} \langle var, rel\rangle$

(DEC-EXPR-VAR) $\quad \langle x, var, rel\rangle \rightarrow_{rel} \langle var, rel[(x, reader) \mapsto rel(x, reader) \cup \{r\}]\rangle$

C.5. Global

$$global\ transitions\ system = (\Gamma_g,\ \Rightarrow_g)$$

$$\Gamma_g = \mathbf{Dec_S} \cup (\mathbf{Rea}^* \times \mathbf{Sto} \times \underbrace{\mathbf{Name_S}}_{curr_s} \times \underbrace{\mathbf{Name_{S\mathcal{E}}}}_{next_s})$$

$$\Rightarrow_g = \Gamma_g \times \Gamma_g$$

(GLOBAL-STARTER)

$$\frac{var, rel, pri \vdash \langle Ds, sto, sta\rangle \rightarrow_{Ds} \langle sto', sta'\rangle}{sta,\ rea \vdash \langle Ds\rangle \Rightarrow_g \langle \emptyset, sto', \texttt{Global}, \varepsilon\rangle} \quad \textbf{where } var, sto, rel, pri, sta = \emptyset \rightharpoonup \emptyset$$

(GLOBAL-REACTION)

$$\frac{\langle c,\ var',\ sto,\ next_s\rangle \rightarrow_l \langle var,\ sto',\ next'_s\rangle}{sta,\ rea \vdash \langle \{..., r, ...\}, sto, curr_s, next_s\rangle \Rightarrow_g \langle \{..., ...\}, sto', curr_s, next'_s\rangle}$$

$$\textbf{where } (c,\ var',\ pri) = r$$

(GLOBAL-EMPTY-TRANSITION)

$$\frac{curr_s \vdash \langle next_s,\ sto, \varepsilon\rangle \rightarrow_t \langle sto', next'_s\rangle}{sta,\ rea \vdash \langle \emptyset, sto, curr_s, next_s\rangle \Rightarrow_g \langle \{r_1,\ r_2\ , ..., r_i\}, sto', next_s, next'_s\rangle}$$

$$\textbf{where } (\{r_1,\ r_2\ , ..., r_i\}, S, pare_s, child_s) = sta(next_s) \textbf{ and } child_s = \varepsilon$$

(GLOBAL-EMPTY-TRAVERSE)

$$sta,\ rea \vdash \langle \emptyset, sto, curr_s, next_s\rangle \Rightarrow_g \langle \emptyset, sto, curr_s, next'_s\rangle$$

$$\textbf{where } (-, -, -, next'_s) = sta(curr_s) \textbf{ and } curr'_s \neq \varepsilon$$

(GLOBAL-MERGE-CONCATENATE)

$$sta,\ rea \vdash \langle \{..., r_1, ..., r_2, ...\}, sto, curr_s, next_s\rangle \Rightarrow_g \langle \{..., r_2; r_1, ...\}, sto, curr_s, next_s\rangle$$

$$\textbf{where } \begin{array}{r} \exists l \in rel(r_2, writer) : l \in rel(r_1, reader) \\ \forall l \in rel(r_1, writer) : l \notin rel(r_2, writer) \cup rel(r_2, reader) \end{array}$$

(GLOBAL-MERGE-INSERT)

$$sta,\ rea \vdash \langle \{..., r_1, ..., r_2; r_3, ...\}, sto, curr_s, next_s\rangle \Rightarrow_g \langle \{..., r_2; r_1; r_3, ...\}, sto, curr_s, next_s\rangle$$

$$\textbf{where } \begin{array}{r} \exists l \in rel(r_2, writer), l \in rel(r_1, writer) \\ \exists l \in rel(r_1, writer), l \in rel(r_3, reader) \\ \forall l \in rel(r_3, writer), l \notin rel(r_1, reader) \cup rel(r_1, writer) \\ \forall l \in rel(r_3, writer), l \notin rel(r_2, reader) \cup rel(r_2, writer) \\ \forall l \in rel(r_1, writer), l \notin rel(r_2, reader) \cup rel(r_2, writer) \end{array}$$

C.6. Transition

$$
\begin{aligned}
transition\ transition\ system = (\Gamma_t,\ T_t,\ \ \rightarrow_t) \\
\Gamma_t &= \mathbf{Names_S} \times \mathbf{Sto} \times \mathbf{Names_{S\mathcal{E}}} \\
T_t &= \mathbf{Sto} \times \mathbf{Names_{S\mathcal{E}}} \\
\rightarrow_t &= \Gamma_t \times T_t \\
& curr_s, next_s \vdash \langle s, var, sto, next_s \rangle \rightarrow_t \langle var, sto, next_s \rangle
\end{aligned}
$$

(TRANS-TOP)

$$
\frac{\langle S, var,\ \ sto, next_s \rangle \rightarrow_l \langle var', sto', next'_s \rangle}{curr_s \vdash \langle s, sto, next_s \rangle \rightarrow_t \langle sto', next'_s \rangle}
$$

where $(-, -, pare_s, -) = sta(s)$ **and** $pare_s \in anc(curr_s)$

(TRANS-TRAVERSING)

$$
\frac{curr_s \vdash \langle pare_s, sto, next_s \rangle \rightarrow_t \langle sto', next'_s \rangle \quad \langle S, var, sto', next'_s \rangle \rightarrow_l \langle var, sto'', next''_s \rangle}{curr_s, next_s \vdash \langle s, sto, next_s \rangle \rightarrow_t \langle sto'', next''_s \rangle}
$$

where $(-, S, pare_s, -) = sta(next_s)$ **and** $pare_s \notin anc(curr_s)$

C.7. Local

$$\begin{aligned} \textit{local transition system} &= (\Gamma_l,\ T_l,\ \ \rightarrow_l) \\ \Gamma_l &= \mathbf{Stmt} \times \mathbf{Var} \times \mathbf{Sto} \times \mathbf{Names}_{\mathcal{SE}} \\ T_l &= \mathbf{Var} \times \mathbf{Sto} \times \mathbf{Names}_{\mathcal{SE}} \\ \rightarrow_l &= \Gamma_l \times T_l \end{aligned}$$

(LOCAL-EMPTY) $\quad \langle \varepsilon,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto,\ next_s \rangle$

(LOCAL-CASE-TRUE)

$$\frac{\langle S,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto',\ next'_s \rangle}{\langle e\ \texttt{:}\ S\ c,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto',\ next'_s \rangle} \quad \textbf{where } e \rightarrow_e \mathit{true}$$

(LOCAL-CASE-FALSE)

$$\frac{\langle c,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto',\ next'_s \rangle}{\langle e\ \texttt{:}\ S\ c,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto',\ next'_s \rangle} \quad \textbf{where } e \rightarrow_e \mathit{false}$$

(LOCAL-STMT-SEQUENCE)

$$\frac{\langle S_1,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var',\ sto',\ next'_s \rangle \quad \langle S_2,\ var',\ sto',\ next'_s \rangle \rightarrow_l \langle var'',\ sto'',\ '' \rangle}{\langle\ S_1\ \texttt{;}\ S_2,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var'',\ sto'',\ '' \rangle}$$

(LOCAL-STMT-TRANSITION-FIRST) $\quad \langle \texttt{enter(}\ s\ \texttt{)}, \ var,\ sto,\ \varepsilon \rangle \rightarrow_l \langle var,\ sto,\ s \rangle$

(LOCAL-STMT-TRANSITION-REPLACE)

$$\langle \texttt{enter(}\ s\ \texttt{)}, \ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto,\ s \rangle \quad \textbf{where } pri(s) > pri(next_s)$$

(LOCAL-STMT-TRANSITION-SKIP)

$$\langle \texttt{enter(}\ s\ \texttt{)}, \ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto,\ next_s \rangle \quad \textbf{where } pri(next_s) > pri(s)$$

(LOCAL-STMT-VAR-DEC)

$$\frac{var, sto \vdash e \rightarrow_e v}{\langle t\ x\ \texttt{=}\ e,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var',\ sto',\ next_s \rangle}$$

where $l = sto(next)$ **and** $var' = var[x \mapsto l][next \mapsto next(l)]$ **and** $sto' = sto[l \mapsto v]$

(LOCAL-STMT-ASS)

$$\langle x\ \texttt{=}\ e\ ,\ var,\ sto,\ next_s \rangle \rightarrow_l \langle var,\ sto[l \mapsto v],\ next_s \rangle$$
$$\textbf{where } var, sto \vdash e \rightarrow_e v \textbf{ and } l = var(x)$$

C.8. Expression

$$\begin{aligned} \textit{expression transition system} &= (\Gamma_e,\ T_e,\ \ \rightarrow_e) \\ \Gamma_e &= \textbf{Expr} \\ T_e &= \textbf{Val} \\ \rightarrow_e &= \Gamma_e \times T_e \end{aligned}$$

(EXPT-NUMERAL) $\quad var, sto \vdash var,\ sto \vdash n \rightarrow_e v \quad \textbf{where } \mathcal{N}(n) = v$

(VAR-REF) $\quad var, sto \vdash var,\ sto \vdash x \rightarrow_e v \quad \textbf{where } v = sto(var(x))$

(EXPR-PARENTHESIS) $\quad var, sto \vdash \texttt{(}\ e\ \texttt{)} \rightarrow_e v$

(EXPR-PIN-READ)

$var, sto \vdash x\texttt{.read()} \rightarrow_e v \quad \textbf{where } v = \text{the value of the pin at } sto(var(x))$

(EXPR-PIN-WRITE)

$$\frac{var, sto \vdash e \rightarrow_e v_1}{var, sto \vdash x\texttt{.write(}\ e\ \texttt{)} \rightarrow_e v_2}$$

where The pin is set to v **and** $v_2 = \textit{void}$

(EXPR-SERIAL-PRINTLN)

$$\frac{var, sto \vdash e \rightarrow_e v_1}{var, sto \vdash x\texttt{.println(}\ e\ \texttt{)} \rightarrow_e v_2}$$

where Write v_1 to the serial connection at $sto(var(x))$ **and** $v_2 = \textit{void}$

(EXPR-PIN-TOGGLE)

$var, sto \vdash x\texttt{.toggle} \rightarrow_e v$

where The pin is set to the opposite possible value **and** $v = \textit{void}$

(EXPR-ADD) $\quad \dfrac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1\ \texttt{+}\ e_2\rangle \rightarrow_e v} \quad \textbf{where } v = v_1 + v_2$

(EXPR-SUB) $\quad \dfrac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1\ \texttt{-}\ e_2\rangle \rightarrow_e v} \quad \textbf{where } v = v_1 - v_2$

(EXPR-MULT) $\quad \dfrac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1\ \texttt{*}\ e_2\rangle \rightarrow_e v} \quad \textbf{where } v = v_1 \cdot v_2$

(EXPR-DIV) $\quad \dfrac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1\ \texttt{/}\ e_2\rangle \rightarrow_e v} \quad \textbf{where } v = \frac{v_1}{v_2}$

(EXPR-EQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ == } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 = v_2$$

(EXPR-EQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ == } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 \neq v_2$$

(EXPR-NOTEQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ != } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 \neq v_2$$

(EXPR-NOTEQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ != } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 = v_2$$

(EXPR-LESSTHAN-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ < } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 < v_2$$

(EXPR-LESSTHAN-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ < } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 > v_2$$

(EXPR-BIGGERTHAN-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ > } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 > v_2$$

(EXPR-BIGGERTHAN-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ > } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 < v_2$$

(EXPR-LESSTHANOREQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ <= } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 = v_2 \vee v_1 < v_2$$

(EXPR-LESSTHANOREQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ <= } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 > v_2$$

(EXPR-GREATERTHANOREQUAL-TRUE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ >= } e_2 \rangle \rightarrow_e true} \textbf{ where } v_1 = v_2 \vee v_1 > v_2$$

(EXPR-GREATERTHANOREQUAL-FALSE) $$\frac{var, sto \vdash e_1 \rightarrow_e v_1 \quad var, sto \vdash e_2 \rightarrow_e v_2}{var, sto \vdash \langle e_1 \texttt{ >= } e_2 \rangle \rightarrow_e false} \textbf{ where } v_1 < v_2$$

(EXPR-NEGATE-TRUE) $$var, sto \vdash \texttt{!}e \rightarrow_e true \quad \textbf{where } e \rightarrow_e false$$

(EXPR-NEGATE-FALSE) $$var, sto \vdash \texttt{!}e \rightarrow_e false \quad \textbf{where } e \rightarrow_e true$$

(EXPR-BOOLOR-TRUE)

$$var, sto \vdash e_1 \texttt{ || } e_2 \rightarrow_e true \quad \textbf{where } e_1 \rightarrow_e true \ \vee e_2 \rightarrow_e true$$

(EXPR-BOOLOR-FALSE)

$$var, sto \vdash e_1 \texttt{ || } e_2 \rightarrow_e false \quad \textbf{where } e_1 \rightarrow_e false \ \wedge e_2 \rightarrow_e false$$

(EXPR-BOOLAND-TRUE)

$$var, sto \vdash e_1 \texttt{ \&\& } e_2 \rightarrow_e true \quad \textbf{where } e_1 \rightarrow_e true \ \wedge e_2 \rightarrow_e true$$

(EXPR-BOOLAND-FALSE)

$$var, sto \vdash e_1 \texttt{ \&\& } e_2 \rightarrow_e false \quad \textbf{where } e_1 \rightarrow_e false \ \vee e_2 \rightarrow_e false$$

D. Type rules

Types

$$
\begin{aligned}
B &::= \texttt{Byte} \mid \texttt{Int} \mid \texttt{Long} \mid \texttt{Float} \mid \texttt{Bool} \mid \texttt{Apin} \mid \texttt{Dpin} \mid \texttt{Serial} \mid \texttt{String} \\
B_v &::= B \mid \texttt{Void} \\
T &::= B \mid x : B \to \texttt{ok}
\end{aligned}
$$

Helping functions:

$$
\begin{aligned}
E(\varepsilon, E) &= E \\
E(s\texttt{,}\ Dp, E) &= E \\
E(\texttt{onenter:}\ S, E) &= E(Do, E[S \to \texttt{ok}]) \\
E(\ T\ x\ \texttt{=}\ e\ Dv, E) &= E(Dv, E[x \mapsto T]) \\
E(\texttt{|}\ e :\ S\ c, E) &= E(Dc, E[S \to \texttt{ok}]) \\
E(\texttt{when}\ c, E) &= E(Dr, E(Dc, E)* \to \texttt{ok})
\end{aligned}
$$

Type rules for e in the abstract syntax. Empty is universal.

(Empty)
$$E \vdash \varepsilon : \texttt{ok}$$

(Num)
$$E \vdash n : T_n \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Var)
$$\frac{E(x) = T}{E \vdash x : T}$$

(Paren)
$$\frac{E \vdash e : T}{E \vdash \texttt{(}e\texttt{)} : T}$$

(Pin-Read-Digital)
$$\frac{E \vdash x : \texttt{Dpin}}{E \vdash x\texttt{.read} : \texttt{Bool}}$$

(Pin-Read-Analog)
$$\frac{E \vdash x : \texttt{Apin}}{E \vdash x\texttt{.read} : \texttt{Int}}$$

(Pin-Write-Digital)
$$\frac{E \vdash x : \texttt{Dpin} \quad E \vdash e : \texttt{Bool}}{E \vdash x\texttt{.write(}\ e\ \texttt{)} : \texttt{Void}}$$

(Pin-Write-Analog)
$$\frac{E \vdash x : \texttt{Apin} \quad E \vdash e : T_n}{E \vdash x\texttt{.write(}\ e\ \texttt{)} : \texttt{Void}} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}\}$$

(Pin-Toggle)
$$\frac{E \vdash x : \texttt{Dpin}}{E \vdash x\texttt{.toggle} : \texttt{Void}}$$

(Serial-PrinIn)
$$\frac{E \vdash x : \texttt{Serial} \quad E \vdash e : T_p}{E \vdash x\texttt{.println(}\ e\ \texttt{)} : \texttt{Void}} \quad \textbf{where } T_p \in B$$

(Add)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{+}\ e_2 : T_n} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Sub)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{-}\ e_2 : T_n} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Mult)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{*}\ e_2 : T_n} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Div)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{/}\ e_2 : T_n} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Equals)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1\ \texttt{==}\ e_2 : \texttt{Bool}}$$

(NotEq)
$$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1\ \texttt{!=}\ e_2 : \texttt{Bool}}$$

(Less)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{<}\ e_2 : \texttt{Bool}} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Great)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{>}\ e_2 : \texttt{Bool}} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(LessEq)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{<=}\ e_2 : \texttt{Bool}} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(GreatEq)
$$\frac{E \vdash e_1 : T_n \quad E \vdash e_2 : T_n}{E \vdash e_1\ \texttt{>=}\ e_2 : \texttt{Bool}} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Neg-Bool)
$$\frac{E \vdash e_1 : \texttt{Bool}}{E \vdash \texttt{!}e_1 : \texttt{Bool}}$$

(Neg-Num)
$$\frac{E \vdash e : T_n}{E \vdash \texttt{-}e : T_n} \quad \textbf{where } T_n \in \{\texttt{Byte}, \texttt{Int}, \texttt{Long}, \texttt{Float}\}$$

(Or)
$$\frac{E \vdash e_1 : \texttt{Bool} \quad E \vdash e_2 : \texttt{Bool}}{E \vdash e_1 \texttt{ || } e_2 : \texttt{Bool}}$$

(And)
$$\frac{E \vdash e_1 : \texttt{Bool} \quad E \vdash e_2 : \texttt{Bool}}{E \vdash e_1 \texttt{ \&\& } e_2 : \texttt{Bool}}$$

Type rules for Dp in the abstract syntax

(Priority list declaration)
$$\frac{E \vdash s : \texttt{ok} \quad E \vdash Dp : \texttt{ok}}{E \vdash s\texttt{, } Dp : \texttt{ok}}$$

(Single state priority)
$$E \vdash s : \texttt{ok}$$

Type rules for Do in the abstract syntax

(Onenter declaration)
$$\frac{E \vdash S : \texttt{ok}}{E \vdash \texttt{onenter: } S : \texttt{ok}}$$

Type rules for Dv in the abstract syntax

(Variable Declaration)
$$\frac{E \vdash e : T \quad E_1 \vdash Dv : \texttt{ok}}{E \vdash T\ x\ \texttt{=}\ e\ Dv : \texttt{ok}} \quad \textbf{where } E_1 = E(Dv, E)$$

Type rules for Ds in the abstract syntax

(State declaration)
$$\frac{E \vdash s : \texttt{ok} \quad E \vdash Dp : \texttt{ok} \quad E \vdash Do : \texttt{ok} \quad E_1 \vdash Dv : \texttt{ok} \quad E_2 \vdash Dr : \texttt{ok} \quad E_3 \vdash Ds_1 : \texttt{ok} \quad E_3 \vdash Ds_2 : \texttt{ok}}{E \vdash \texttt{state } s \texttt{ \{ } Dp\ Do\ Dv\ Dr\ Ds_1 \texttt{\} } Ds_2 : \texttt{ok}}$$
$$\textbf{where } E_1 = E(Do, E) \textbf{ and } E_2 = E(Dv, E_1) \textbf{ and } E_3 = E(Dr, E_2)$$

Type rules for Dr in the abstract syntax

(React)
$$\frac{E \vdash c : \texttt{ok}}{E \vdash \texttt{when } c : \texttt{ok}}$$

Type rules for c in the abstract syntax

(Case)
$$\frac{E \vdash e : \texttt{ok} \quad E \vdash S : \texttt{ok} \quad E_1 \vdash c : \texttt{ok}}{E \vdash \texttt{| } e \texttt{ : } S\ c : \texttt{ok}} \quad \textbf{where } E_1 = E[S \to \texttt{ok}]$$

Type rules for S in the abstract syntax

(Sequence) $$\frac{E \vdash S_1 : \mathsf{ok} \quad E_1 \vdash S_2 : \mathsf{ok}}{E \vdash S_1 \texttt{;} S_2 : \mathsf{ok}} \quad \textbf{where } E_1 = E[S_1 \to \mathsf{ok}]$$

(Assignment) $$\frac{E \vdash x : T \quad E \vdash e : T}{E \vdash x \texttt{ = } e : \mathsf{ok}}$$

(EnterState) $$\frac{E \vdash s : \mathsf{ok}}{E \vdash \texttt{transition(} s \texttt{)} : \mathsf{ok}}$$

(Statement-Expression) $$\frac{E \vdash e : T_v}{E \vdash e : \mathsf{ok}} \quad \textbf{where } T_v \in B_v$$

Type rules for p in the abstract syntax

(Program) $$E \vdash Ds : \mathsf{ok}$$

E. Code Examples

E.1. DigitalReadSerial Example - Arduino Language

```
1  int pushButton = 2;
2
3  void setup() {
4    Serial.begin(9600);
5    pinMode(pushButton, INPUT);
6  }
7
8  void loop() {
9    int buttonState = digitalRead(pushButton);
10   Serial.println(buttonState);
11   delay(1);
12 }
```

E.2. DigitalReadSerial Example - Rios

```
1  dpin pushButton = @ 2
2  serial sout = $ usb 9600
3
4  every 1ms : sout.println(pushButton.read())
```