Pivot

- The Language for Home Automation -

Project Report
D410f19

Aalborg University
Computer Science

Hannah Lockey

Asger Gammelgaard Bertel

Philip Irming Holler

Magnus Kirkegård Jensen

Mads Schou Faber

Mads Kristian Bau-Madsen

Computer Science
Selma Lagerløfs Vej 300
9220 Aalborg Ø
http://www.cs.aau.dk/

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Pivot - The language for home automation

Theme:
Computer Science, Domain Specific Language

Project Period:
Spring Semester 2019

Project Group:
D410f19

Participant(s):
Asger Gammelgaard Bertel
Philip Irming Holler
Hannah Lockey
Magnus Kirkegård Jensen
Mads Schou Faber
Mads Kristian Bau-Madsen

Supervisor(s):
Michele Albano

Copies: 1

Page Numbers: 112

Date of Completion:
May 23, 2019

Abstract:

This project concerns itself with the design of a programming language and the implementation of the compiler that is needed for the programming language to be compiled. Pivot is a domain specific programming language that is designed to make it easier for programmers to implement their own IoT programs. During the problem analysis and language design phase, it was put to light, that essential tools were missing in the currently existing languages. General purpose languages require too much boilerplate code, leading to inefficient use of time. Throughout the project, the semantics for Pivot has been updated simultaneously with the implementation, and there has been much emphasis on making sure the implementation corresponds with the semantics shown in section 5. The compiler is tested with unit tests, system tests, and the programming language is tested with the use of a client server program sending data to each other using the pivot code. The programming language has yet to be tested on real IoT hardware, but changing the protocol and writing drivers, is relegated to future work.

Contents

Abstract

This project concerns itself with the design of a programming language and the implementation of the compiler that is needed for the programming language to be compiled. Pivot is a domain specific programming language that is designed to make it easier for programmers to implement their own IoT programs. During the problem analysis and language design phase, it was put to light, that essential tools were missing in the currently existing languages. General purpose languages require too much boilerplate code, leading to inefficient use of time. Throughout the project, the semantics for Pivot has been updated simultaneously with the implementation, and there has been much emphasis on making sure the implementation corresponds with the semantics shown in section 5. The compiler is tested with unit tests, system tests, and the programming language is tested with the use of a client server program sending data to each other using the pivot code. The programming language has yet to be tested on real IoT hardware, but changing the protocol and writing drivers, is relegated to future work.

Preface

This report is written by six 4th semester computer science students from Aalborg University in the period from February 1st to May 24th. The project assignment involved designing, defining and implementing a new programming language using syntax and semantics, as well as using implementation techniques learned in the courses in parallel with the project process.

Chapter 1

Introduction

Connectivity and communication between household electronic devices has in the past years become more popular[12], and to describe the network of various devices in our homes, the term "Internet of Things" (IoT) is used. Smart devices such as TVs, thermostates, lamps, Google Home and various other devices are being developed and distributed to regular households, and with more household devices becoming intelligent, the potential of IoT also becomes greater. Currently, the software for IoT is primarily being programmed in general purpose programming languages such as Java, C++ or Python[12][14]. The purpose of this project is to create a specialised domain specific language for IoT development, to enable easier programming and customisation of smart homes systems.

Chapter 2

Problem Analysis

In this chapter the problem will be described in detail. This leads to a problem description which will be the basis for the solution. The initial idea started, when it was found out, that home automation software was primarily written using a general purpose language[14]. This begged the question of whether or not the process of writing a home automation system could be made easier and or faster with a domain specific language.

In order to gain a better understanding of home automation, we will begin by describing the meaning of the term. Home automation is defined as a system that automates some household tasks such as turning lights on and off, regulating heating systems or interacting with household appliances [23]. Systems might employ time based events to control devices. Such systems are typically provided by the manufacturers of the smart-devices, and allows users to define their own home automation events, within the scope of the pre-programmed control software. Using a programming language to define rules and events could widen the scope, and allow the user a greater degree of freedom when creating home automation systems.

Currently there are several existing systems for managing home automation. Some popular systems include for example the smart lightning systems called Philips Hue[21] and IKEA Trådfri[13]. Other systems for home automation include Siemens connect, which allows smart managing and automation of home appliances such as refrigerators and washing machines. These systems enable consumers to automate their homes by, for example, using a timer to control lights. The systems, however, require the user to use software either written by these companies or written using API's from these companies. Also the software will for the most part only work with hardware from that specific company. This kind of specialised hardware might be expensive and or not fitting the specific need from the consumer. For example a regular Philips Hue Bulb for the E26 socket costs

29.99USD[21]. This means that a cheaper and more flexible solution could be desired by some users.

The stakeholders for this project, would be not only the users wishing to create software for IoT, but also potentially the producers of hardware and software used in home automation. The best case for this project would be, that the development of IoT software through our programming language, can be applied to all kinds of devices without any change in hardware, but this is fairly optimistic. Furthermore, the stakeholders that can influence this project, would be the ones producing products that are the target devices for our programming language, due to them controlling the procedures for communicating with the devices.

Most current home automation programs are written in general purpose languages such as Java, Python, C and C++ [14]. Except for Python, all of these languages are of the C-family of programming language, meaning that they have many similarities in syntax. All of these languages are general purpose languages, which can lead to some solutions having many lines of unnecessary coding, which can increase the code complexity and the needed time to develop and test it. For example in Java, if a programmer wanted to create an event based program, he or she must first create either an infinite loop inside the main method or listeners for events. This main method would then also need to be in a class, since java is strictly object oriented. Both these things add lines of code, that does not actually contribute to the functionality of the program. A programmer trying to implement an event based home automation program to manage domotic devices, would need to create an event handler and possibly a queue for sorting the events after time, depending on the specific implementation. This will further increase the amount of programming needed, even though these things will probably stay the same for most implementations. Some of this could also be solved using some libraries for Java, C# etc, but this would still lead to more work to design, implement and debug these.

The extra lines of code and additional programming leads to the need for a higher level language, that would automatically take care of timing, queue, loops, listeners etc. This would free up time for the programmer to focus on implementing the logic for the events. In this report we will limit our research to event based programming as a solution.

Another problem with the current solutions for home automation is, that there are many different ways of communicating through the network to the smart devices. In some cases, proprietary software and hardware is used for this purpose. This leads to a fragmentation in the market between the different brands and approaches. For this reason a new higher level programming language would also

either need to compile to a program with predefined communication protocols for some of the current standards, if possible, or some structures inside the language to customise these protocols. This would either add support for current hardware by customising an existing protocol, or create a new standard, that the hardware then needs to be adapted to.

2.1 Existing Solutions

In the search for existing systems we limit our search to programming languages for implementing home automation. We do not consider, for example the IKEA Trådfri software a competitor, since it is not a way of creating home automation software, but instead a software for managing an existing smart lightning system with only IKEA hardware.

Tools such as Calaos[2] and MisterHome[16] already exists for creating custom home automation solutions for a variety of devices. Both of these tools limit themselves entirely to configuration through a user interface with a limited amount of predefined customisation options. This again distances them from a programming language for home automation.

Domoticz is a tool which allows further customisation using a drag and drop programming solution[7]. Domoticz also allows adding new types of devices by defining which signals they can receive, there is, however, only a limited amount of signals to choose from and no way to define new signals. This limits the amount of devices that the system can interact with. [7]

Home Assistant is an interface-based tool like Calaos and MisterHome, but it also allows scripting in python [9]. But python is a general purpose language which, as previously discussed, may lead to suboptimal programming when compared to a domain specific language[22]. Similarly OpenHAB[18] allows scripting, but it has its own domain-specific programming language. One downside to OpenHAB is that it only works with a set of predefined devices.

Another group at Aalborg University created a 4th semester project regarding a programming language for home automation in 2014[5]. This group implemented a language called Home, that could be used to create event based home automation programs, that take input from hardware to trigger an event. They did, however, not manage to implement a timer for their home automation. This means, that an event cannot be triggered at a certain time or date, but instead only when some hardware sends an input to the program. The focus of the Home language was also to enable novice programmers to code their own home automation software using a simple high level language.

It appears that existing languages for home automation limit the user, either by

only supporting a predefined set of devices, or by lacking crucial features such as timed events.

2.2 Problem description

The problem analysis led to the following problem description which will be used for creating a solution to the problem.

- **Problem:**
  How can a higher level language be designed, that enables easier development of an event based program to control home automation?
  - **Subproblems:**
    How should this language be compiled to run a server device?
    - How would the compiled program communicate with the smart devices?
  - How should the syntax look in order to optimise the code for the specified user?
    - Who would be the user of this language?
    - Language evaluation criteria

2.3 Solution criteria

In order to make sure that the proposed language can be considered a solution, a number of criteria has been made. These will be used to evaluate the final language. Additionally they will be used to conclude which characteristics are the most important for the language in section 3.3. The following criteria has been established based on the problem analysis:

- The language must enable the user to easily express and solve home automation problems.
- The language must enable the user to control devices based on incoming signals and time-based events.
- The language must support addition of new types of devices.

The first criteria is very broad, but serves as a guideline for designing and creating the proposed language. The first criteria reflects on the ease of use as well as how easy it is to understand, specifically in relation to home automation. This is important, since being able to easily write and understand the code, will allow developers to be more productive. This could for example be measured by the

amount of boilerplate code. The second criteria revolves around the requirement to allow interaction with sensors and enable time-based events to be created. The third requirement specifies that the language must enable support of new devices that are not supported by default. This could be accommodated by allowing the user to define basic custom communication protocols.

Chapter 3

Design Decisions

Now that the problem is well defined and described, see chapter 2, some design decisions must be made. These decision must, however, be based on an analysis of the domain in which the solution will function. This chapter includes and domain analysis which then leads to an analysis of which qualities would suit this domain. after this the target language is chosen based on the domain analysis. The target language then affects the parsing tool choice, which is also discussed at the end of this chapter. But first, the name of the domain specific language is described.

3.1 Language name

The language described in this report will be called Pivot. A program for home automation is supposed to be the pivotal point of a smart home. For this reason the name Pivot was chosen. For the rest of the report Pivot will refer to the language being developed.

3.2 Domain Analysis

This section will more specifically define the domain which the Pivot language will be specific to. This includes a user analysis, a specification of which hardware the language aims to run on, as well as a short example of what a potential system could look like.

3.2.1 User analysis

The language should be tailored to a user that not only has an interest in smart devices, but also has the expertise and knowledge to even consider developing his/her own solutions, instead of simply using existing solutions such as the

aforementioned Philips Hue[21]. According to a survey by Statista, there are currently 6.7 million households in the UK that can be considered "Smart Homes"[26]. There are around 27 million households in the UK[17], meaning that almost 25% of households in The UK are "smart" in some way. This, however, does not mean that there are 6.7 million potential users of Pivot, since a reasonable grasp of general programming concepts will likely be required to use it. Looking at similar solutions, such as openHAB, mentioned in section 2.1, the userbase is much smaller[6]. While there is an increase in interest in smart homes, the userbase for a DSL will always be limited to people with the necessary programming knowledge. The Pivot language will therefore be developed for a user with some programming experience. As described in section 2.2 most current solutions are implemented in Python, C, C++ or Java. Since all of these languages, except for Python, are based on the C syntax, Pivot could reuse some of the syntax from these languages to make the language more familiar to the user.

3.2.2 Hardware specification

The language is intended to make the programming of home automation simpler and more effective. Pivot should not control how the devices react to the signals being sent to them, but be a central server controlling input and outputs from all smart devices in the home. Signals are the messages that the server and client sends between each other to communicate. They consist of data, which will be sent with a predefined protocol. The hardware devices are themselves responsible for reacting correctly to the signals. For this reason the language will be designed with a client/server structure in mind. This means that the program written in Pivot will run on a central server and send out and receive messages from the smart devices. This approach is similar to the one taken by for example IKEA with their home automation hardware. IKEA sells the TRÅDFRI Gateway to be used as the hub for the entire system[13]. The server running the pivot language program is connected to all the sensors, light bulbs and other devices. An example of a system is showcased Figure 3.1

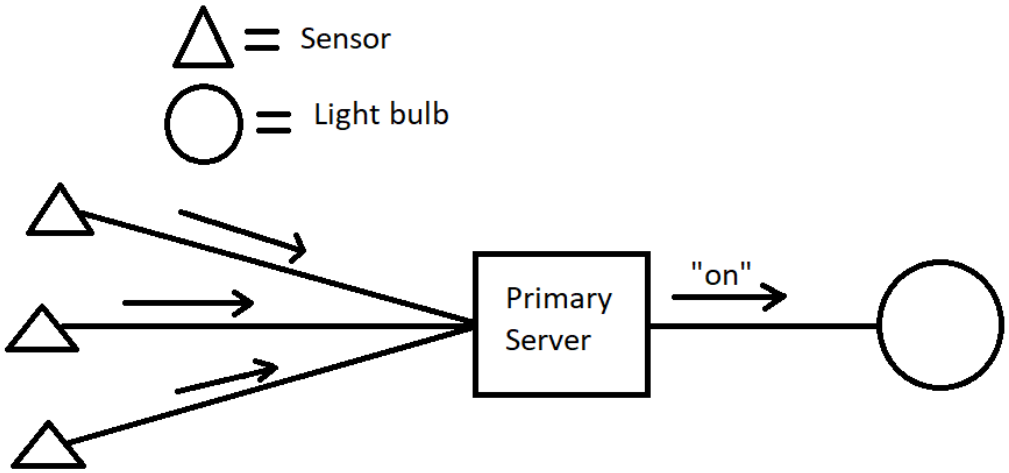

Figure 3.1: An image of what a system, that the proposed language could support, could look like

The sensors send data to the server, which then analyses their output and determines if it should trigger an event. In the illustration, the outcome of the output data is that a light bulb should be turned on. The language should therefore be designed with this kept in mind. While this example system is very simple, the language should also support any appropriate kind of sensor or smart device.

3.2.3 Server/Client Protocol

Since the language is intended to use a direct client/server communication, a clear communication protocol must be developed. The messages sent in between the devices and the server will all consist of Strings and all messages will consist of three separated by a space. The reason for this is each word has its own meaning. When a device connects to the server, it will have to send information about itself, in order for the server to know the identity of the device. In our case, it consists of a hardware-ID, an IP Address and port number. After the initial message whenever a device and the server exchanges messages, the latter contains a Hardware-Id, Type and Data. Hardware-ID is a unique id that each device has, which is used to differentiate between devices on the network. Type specifies the data type that the device uses. For example a lamp receives a Toggle signal and a temperature sensor sends Celsius signal. Data depends on the type, for instance, if the type is Toggle the data can either be "On", "Off". If the type is Celsius the data will be a number. This means that a message could be: "1 Celsius 25", where "1" is the hardware id, "Celsius" is the signal type and "25" is, in this case, the temperature. For this reason the programmer must be able to define signal types and what kind of data these types can hold in the Pivot program.

3.3 Language Qualities

Now that the domain of the Pivot language is known, it can be determined which qualities and domain specific language to this domain should hold. In this section, the qualities that the language should have will be described and argued for. This will be based on the language criteria described in the book "Concepts of Programming Languages" by Robert W. Sebesta[24].

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Figure 3.2: Language evaluation criteria and the characteristics that affect them

First it will be analysed which of the criteria are most important for the language Pivot. Then it will be analysed which characteristics will help create a language, that can fulfil these criteria.

Criterion	Very important	Important	Less important
Readability		X	
Writability		X	
Reliability	X		

Table 3.1: Criterion table

Since the Pivot language will most likely be used by fairly experienced users the readability and writability criteria will not be the most important. It is assumed, that not many people, who are new to programming, will try to automate their home with a piece of software written in the Pivot language. Experienced programmers will most likely have seen more complex programs in other languages. Reliability, however, will be very important. Since smart homes are unreliable environments, both in terms of communication being unreliable and the stability of

connected devices. Since writability and readability affects the ease of maintenance of a program, they are also important in order to achieve a higher level of reliability.

3.3.1 Readability

When judging a programming language one of the most important criteria is readability[24]. If a program is readable it makes it easier for other programmers to understand, continue writing and maintain the program. For Pivot, readability is marked as important in table 3.1. Therefore simplicity is an important criteria, since a smaller amount of basic constructs will make the language easier to learn[24]. Another way to increase readability is to make the language more orthogonal. Orthogonality means that primitive constructs can be combined in a certain amount of ways. The more ways these constructs can be combined the less exceptions will be caused and therefore the code will be easier to read. An example of this could be using the + operator between two strings to concatenate them. However, too much orthogonality can cause a negative impact on the readability, since a much larger amount of combinations of primitive constructs can become more complicated to read. It is therefore important to not overemphasise on orthogonality. Another way to increase readability is to have a proficient way to define data types. In C Boolean variables does not exist, which means you have to define true and false as non-zero and zero.

3.3.2 Writability

Programs written in Pivot should enable the user to more easily express and solve common home automation problems, than general purpose languages such as Java or Python allow for. According to Robert W. Sebesta, writability is strongly connected to expressivity [24]. Expressivity concerns being able to specify computations in a convenient way, where it is intuitive and logical to write code in the program using the experience from similar language syntaxes. The aim for Pivot should therefore be to increase expressivity specifically in relation to home automation. This could for example be a keyword *now*, that reflects the current time a run time. Since the Pivot programming language would revolve a lot around time based events, some syntax for easily writing and manipulating time should be included.

3.3.3 Reliability

According to Sebesta[24] there are several aspects that can affect reliability. In chapter 3.3 it was decided that the Pivot language should have a high emphasis on reliability. For this reason type checking is important. Type checking should either be done at compile time or run time, where compile time would be more desirable

due to the higher performance[24]. If a language is statically typed, all type errors can be discovered at compile time[10]. This property should be included in the Pivot language to increase reliability. Another way to increase reliability would be exception handling. This ensures that a program will behave in a well defined way in case of run time errors. In order to increase reliability aliasing should not be a thing in Pivot. This means that pointers should be avoided, since they allow for two variables to access and modify the same memory space. In addition to this, some mechanisms for avoiding deadlock and other problems related to concurrently running events should be implemented.

3.4 Target Language

Now that qualities, that the Pivot language should have, have been found, a way of running the programs should be found. How should the Pivot code get from a state of Pivot code to some machine instructions, that a computer, which is likely to be in the domain, can execute. It has been decided to compile the Pivot language program to Java. In order to execute Java code on a machine, it is first compiled to Java ByteCode using the JavaC compiler. After this the ByteCode is compiled using a JIT compiler to machine code local to the machine running the Java Runtime Environment. With this setup, a compiled Java program in Java Byte Code can be executed on all machines, that are support by the Java Run Time Environment, due to the intermediate code, ByteCode, being system independent. This means, that the Pivot program compiled to Java can run on many different machines, including popular architectures like x86 and ARM.

Another good reason for choosing Java as the target language is that it is one of the popular general purpose languages currently used for writing home automation software[14]. This means that programmers using the Pivot language will likely also understand and be able to manually modify the output program written in Java. C#, C++, Python and other general purpose languages would also have been good options, but Java was chosen due to the authors of this report having more experience with Java.

In order to visualise how a Pivot program would get to a state of Java a tombstone diagram was created. If the Pivot program gets compiled to ByteCode, it will be able to run on almost all machines, which increases the versatility of the language.

In order to get the Pivot code compiled to Java, or indeed Java ByteCode, the compiler must be able to run. The compiler for Pivot is also written in Java. To get this Pivot compiler running, a compiler-compiler is needed. In the case of the Pivot language, the compiler-compiler is the JavaC compiler, that compiles from Java to ByteCode. As seen in figure 3.3, this results in a Pivot compiler in written in ByteCode, that can then run in the Java Virtual Machine.

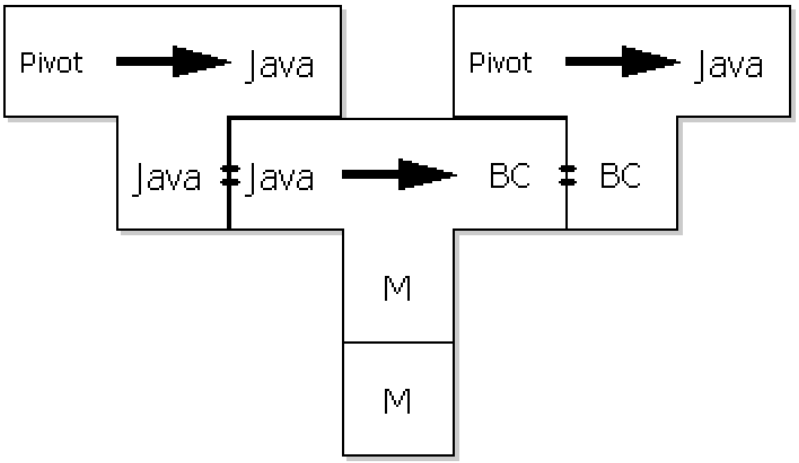

Figure 3.3: Getting the Pivot compiler compiled to ByteCode

Now the Pivot compiler can be run on machines to compile Pivot programs to Java. This process is shown in figure 3.4. The Pivot program is compiled using the compiled compiler constructed as seen in figure 3.3.

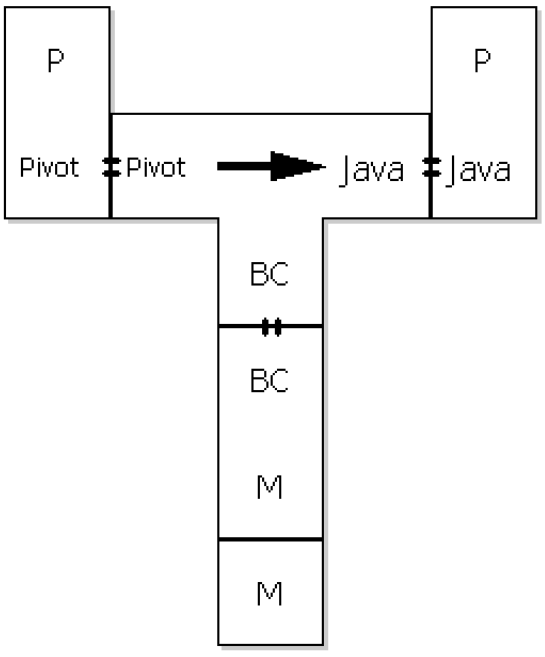

Figure 3.4: Using the compiled Pivot compiler to compile Pivot programs

This analysis shows, that it is possible to get from the Pivot code to the target code using the Pivot compiler. This target code is also able to run on machines, that are popular in the domain, where the language will be used.

3.5 Parsing Tools

The Pivot compiler is written in Java and was created partly by using a parser generator tool. The decision to write the compiler code in Java was based on the available libraries for generating parsers and the fact that the group members are already experienced in Java. Additionally, Java's object-oriented paradigm allows for convenient representation of the different nodes in the parse tree and the abstract syntax tree, both of which are essential parts of the compiler.

When creating a parser, several decisions have to be made. First, which parser variant is needed, for example LL(1) or LL(*). LL(1) is a very fast and easy to implement solution in the form of a recursive decent parser. It is, however, also limited. For example it cannot parse rules with common prefixes, since it cannot immediately determine which of the rules to use, which introduces ambiguity. Since Pivot will be using different rules for different events, that all start with the keyword *when*, an LL(1) parser cannot be used.

Manually implementing a more powerful parser can be a complex task. For this reason, and due to the extensive array of parser tools, it was decided to look for a fitting parser tool.

Pivot uses the ANTLR parser generator, but there exists several other java libraries for compiler construction such as JavaCC, CUP, SABLE CC and several more. Many of these were considered, but only the two primary candidates ANTLR and JavaCC will be described in detail here.

Both ANTLR and JavaCC can generate a parser in Java. ANTLR generates an LL(*) parser, where JavaCC generates a LL(k) parser. Both are able to handle a common prefix. LL(k) for example does it by having a k number of lookahead, which then determines the correct rule to use.

The grammar specification for ANTLR is written in a format very similar to EBNF. Additionally, ANTLR has extensive support for IDEs such as IntelliJ, which is frequently used for Java development. The ANTLR plugin for IntelliJ features a tool, which brings various features to enhance workflow such as syntax- and error highlighting for ANTLR grammar files, as well as code completion when editing the grammar. The plugin also allows the user easily to generate the parser source code in Java from within the IDE.

The ANTLR plugin can also generate and visualise parse trees for a grammar in real time. This allows the user to see the parse tree for the grammar by provid-

ing a sample Pivot program. The parse tree also visualises any lexical errors that might occur and is updated in real time when the grammar or the sample pivot code is changed. This allows for fast debugging and development.

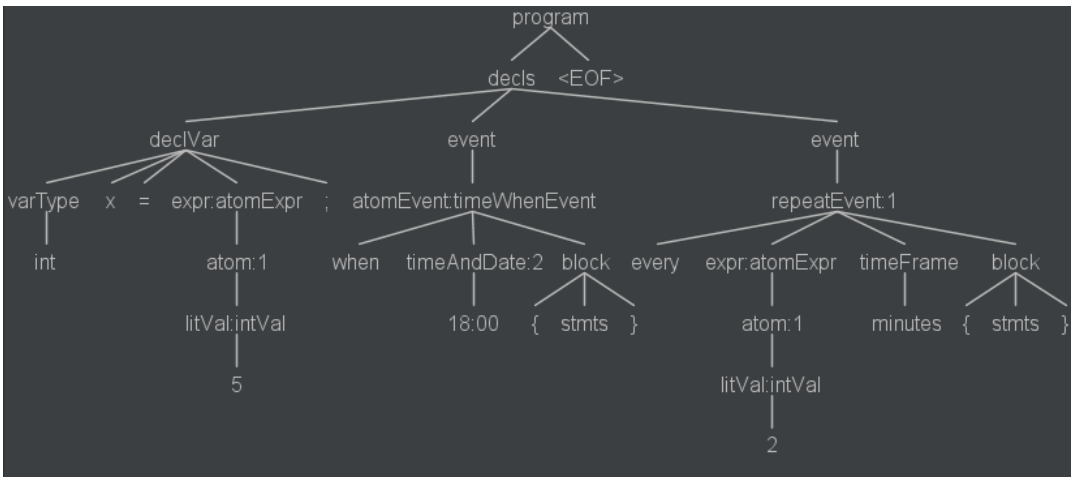

Figure 3.5: Parse tree generated by the ANTLR IntelliJ plugin

JavaCC is another java parser generator. Grammar specification within JavaCC resembles regular Java code more so than EBNF. JavaCC also has plugins for various IDEs, but the functionality is not as extensive as the ANTLR plugin. The plugin mainly provides error- and syntax highlighting for JavaCC files. Unlike ANTLR, the parser code can only be generated by launching the parser generator externally through a terminal. JavaCC compilers seem to be considerably faster than ANTLR-based compilers, but since speed is not really a concern for the Pivot language, this did not factor in greatly when choosing a parser generator [4].

Ultimately ANTLR had several advantages such as extensive IDE support and real-time parse tree visualisation which led us to use ANTLR for creating the Pivot compiler.

3.6 Summary

The Pivot language will be developed for a client/server hardware environment, where the program written in Pivot will be run on a central server. This server machine will then send and receive commands from the home automation devices. The user base of the language will likely be composed by fairly experienced programmers. The most important quality for Pivot is reliability. Readability and writability are also important, but less so due to the users likely being fairly experienced programmers. The target language of Pivot will be the Java programming language. The Pivot language will be translated from Pivot to Java by using a

parser created with the ANTLR parser generator tools combined with a compiler written in Java.

Chapter 4

Language Design & Syntax

In this chapter it will be discussed how the Pivot language syntax will be designed. Initially, some of the knowledge gathered in the previous chapters will be used to briefly describe Pivot. Following this, an example program will be discussed with reasoning for the syntax.

4.1 Initial Description of Pivot

Pivot is an event based programming language, which means that the user defines events, that are executed when a defined boolean expression evaluates to true. These booleans can consist of both signals being processed or a time. The program runs through the events, and when the guard for an event is true it runs that event. Pivot is mostly event based except for an *init* function that executes as soon as the program starts. The init function cannot take parameters or be called by other functions. This function is introduced to enable setup messages for devices in case, these are needed. The events will not run before the execution of the init function is done. Pivot introduces two unique data types called *Device* and *Signal*. The Signal type is for declaring which types of signals a device can send or receive. A Device can act as an actuator, a sensor, or both. In the first case, the Device can receive commands. In the second case, it can provide data to the server. When instantiating a device one must first define the device type, for instance a Bulb, and afterwards what type of signals the device type supports as either input, meaning sent from the server to the device, or out, meaning output from the device to the server. Following this, it is possible to declare an arbitrary number of devices with the type Bulb, each with a unique ID. These all have support for the signals associated to the device Bulb. When a device is instantiated it is assigned a string. This string can then contain for example the IP address of the device on the network. The language also contains functions that can only be called within events or the init function. To make it possible for the programmer to delay the execution of some

actions, Pivot contains a *wait* method which delays the thread for however long the programmer wants.

The idea behind Pivot will lead to a pivot program having the patteren represented in code snippet 4.1.

```
1 define Devices
2
3 define Signals
4
5 declare devices
6
7 initFunction
8
9 functions and events
```

Code snippet 4.1: Pivot pattern program

4.1.1 Keywords

To create an overview over the keywords used in Pivot, this section will shortly describe the functionality and syntax used for the unique keywords, as well as providing an overall list of reserved keywords in Pivot, as seen below.

- Signal
- input
- output
- Device
- define
- if
- while
- init
- void
- when
- every
- ms
- seconds
- minutes
- hours
- days
- weeks
- months
- wait
- string
- int
- float
- set
- get
- exceeds
- deceeds
- now
- else
- return

Many of these keywords, such as *if* and *while*, are commonly used in other programming languages. Table 4.1 describes the keywords that are unique to the Pivot language. The rest of the keywords, that are also present in Java, will have the same behaviour as in Java.

Keyword	Description
when	The when keyword is used in event declarations for defining the condition that must be fulfilled to trigger an event. This means, that **when** *something* happens, the event is run. For example "when 14:00" would trigger every day at 14:00 o'clock. Or "when frontDoorLight toggle : On" meaning when the device frontDoorLight's signal toggle is On.
every	The every keyword is used to specify events that are triggered by a time-based condition, such as **every** *3 seconds*. Generally the every keyword has the following syntax: **every** *time*, where time can be in the following types: **months**, **weeks**, **days**, **hours minutes**, **seconds** and **milliseconds**. When the statement ***every** 3 **seconds*** is made, the program will wait 3 seconds from the execution start and then run the event every three seconds.
define	Whenever the user wishes to define a new signal or device, that the system is connected to, the user has to use the keyword **define**. This is to create more structure, and make it clear that a new signal or device has been defined for the sake of readability. These defined signals and devices are similar to classes in Java.
Signal	With the **Signal** keyword a signal can be constructed. A signal includes signal literal values or ranges of values to describe their state. The signal literals act similarly to enums in Java.
Device	The **Device** keyword is used to define devices. A device can use any number of signals as both input and outputs.
input/output	An **input** or **output** describes what signal the device sends out, or what kind of signal the device wishes to receive when communicating with the server. A device can have multiple outputs or inputs.
exceeds and deceeds	**exceeds** and **deceeds** are used as boolean expressions. When a signal value exceeds/deceeds a defined value, they will evaluate to true. For instance, if a Signal outputs a value that exceeds the defined value, it will evaluate to true. However, it will not be able to evaluate to true unless its goes under the defined threshold again.
now	The now keyword is made to reference the current time. Now can only be used inside events.
set	set returns the value equal to the signal sent to a device. Set also saves the sent signal value in the background.
get	get retrieves the last value that was sent to a device if the signal was input. If the signal was output the get retrieves that last signal that was received. If there has been no signal output or input from a device, get returns lower bound if the signal has a range or else the int value 0.

Table 4.1: Descriptions of keywords in Pivot

4.1.2 Data types

Pivot contains 3 built-in data types: int, float and string. Both ints and the floats are identical to the java implementation. The int is a 32-bit integer value and the float is a 32-bit floating point value with approximately 6-7 significant decimal digits. Unlike Java, string is a keyword and has no methods attached to them.

Pivot uses static typing. This means, that all variables must be typed, and no variable can change type or be cast to another type. The reason why strong typing was chosen is, that this enables type checking at compile time, which in turn can increase reliability at run time. This was a goal according to the language quality analysis at 3.3.

Pivot uses static type binding where the type is determined at compile time and does not change throughout the execution of the program. Additionally, the variable types are determined explicitly, meaning that the user must specify the type of the variable when declaring it. Variables must always be given a value upon declaration since there are no default values.

In Pivot all variables must be initialised, when they are declared. This is used to increase reliability in accordance to the language quality analysis 3.3. In Pivot it is therefor not possible, that some undefined behaviour can be caused by an expression with a variable, that is not initialised, but is declared.

Pivot has some data abstractions. This comes in the form of type defining of signals and device types.

In the first secontion of a Pivot program, signals are defined. Signals can contain either some signal literals or a range of ints or floats. The signal literals work similarly to enums in Java. Each signal literals is set to be a value, and when the name of the literal is used, the value is inserted instead at compile time. This is used to increase readability, which in turn can increase reliability. For example one can write *set bulb1 Toggle: On* to turn on the bulb. This can be more readable than *set bulb1 Toggle: 1*. The ranges are used to make sure, that the device can only take valid input when it accepts ints or floats. For example a door opener can only open the door from zero to one hundred percent. This can decrease the risk of undefined behaviour in the devices at run time, which increases reliability of the program.

The device types then implements the signals as either input or output. An input means, that the device can take input from the server of this signal type. Output means, that the server can expect signals of this type from the device to the server. The device type will then have getters for both input and output signals and then setters for input signals.

When devices are declared, they must have a device type as their type. This type then enables behaviour such as get and set for the correct signals as described above. When the devices are declared, they are initialized using a string. This can then for example contain some IP for the device on the network.

4.1.3 Variable Scope and Lifetime

Pivot uses static scoping where variable scopes are determined at compile time. Pivot programs are organised in blocks attached to events, functions or control structures. Any variables declared outside a block is a global variable which is directly accessible anywhere in the code. Local variables are variables declared inside a block and these variables are only accessible within this block and in any blocks that might be nested within it. Pivot makes use of the java garbage collector for deallocating unused variables. This is described in section 4.1.4.

4.1.4 Garbage Collection

When Pivot deallocates unused variables it uses Java garbage collection which works by looking at the heap memory and deleting unused variable. An unused variable is a variable in the heap that the program no longer has a reference to or points to.

4.1.5 Events

To create events, we decided to include two keywords, making it easier for the user to control when the events should be triggered. This is done with the keywords **when** and **every**. When is used for single execution events, whereas every is used for events, that trigger with an interval. Furthermore, only a few booleans (**exceeds**, **deceeds** and time based) can be used in the event declaration. When an event is triggered while already running, the previously running instance will be terminated. This ensures that the server can at most have as many threads running as the program has events. This characteristic is instrumental to guarantee the language's reliability.

4.1.6 Event lifetime

When the conditions for an event are fulfilled, a new thread is created which runs the code inside the event declaration. Exactly one thread can exists for each event declaration. In the case where another thread from the same event is already running when the event is triggered, the old thread is terminated. This ensures that there can never multiple instances of a single event running at the same time, interfering with each other. Furthermore, by interrupting the running thread, we prevent an non-terminating event from blocking all future executions of that event. This can increase the reliability of the program, which was a goal according to the analysis in section 3.3.

4.1.7 Function Parameters

In Pivot the actual parameters in a function call must match the formal parameters in both order and type. The parameters are then passed by value. The reason for this is again to increase reliability, since all variables inside a function from the formal parameters then has a value. Pass by value was chosen, since Pivot already has global variables. Also in Pivot all devices are declared in the global scope, meaning that they can be accessed from any scope. For this reason it is not necessary to pass an object by reference. This same functionality can be achieved by declaring the variable in the global scope and changing it in the function.

4.1.8 Precedence for operators

Since the pivot language compiles to java the precedence will be the same as java. This means the language uses ordinary mathematical precedence like multiplication before addition and with parentheses the programmer can create their own precedence.

Pivot's associative structure like precedence follows javas and can be seen in the table below.

Precedence	**Operator**	**Associativity**
1	*, /	left-associative
2	+, -	left-associative
3	<, >, <=, >=	non-associative
4	==, !=	left-associative
5	&&	left-associative
6	||	left-associative

Table 4.2: Precedence and associativity in Pivot

Precedence rules can be avoided by allowing for parentheses, which is also a possibility in the Pivot language. This enables the programmer to decide on precedence. This means that an expression can simply be evaluated left to right or right to left, see [24].

4.2 Pivot program example

In order to show how a program, that solves the problems described in the problem analysis in Section 2, would look like, a number of program examples were made. In this report however, only one will be showcased. The examples allow for finding a way to express a solution to a given problem in the Pivot language. This is also useful for finding the abstract syntax of the Pivot language.

In order to come up with code examples, some use cases were first thought out. The code in code snippet 4.2 shows the following use case.

1. A user wants the lights to turn on at the frontdoor when someone approaches the house. This should, however, only happen in the evening where it is dark.

2. The user also wants a cool bedroom, when he/she comes home from work. This can be achieved by opening the window in the bedroom window.

3. In the summer the user wants the lawn mower to run automatically every two weeks, but only in the summer months between 1st of April and 30th of September.

```
// Define signals
define Signal toggle: on = 1, off = 0;
define Signal openWindow: 0.0..100.0;
define Signal celsius:   50..70;

// Define devices
define Device Bulb input: toggle;
define Device Window input: openWindow;
define Device LawnMower input: toggle;
//define sensor devices
define Device MovementSensor output: toggle;
define Device Thermometer output: celsius;

// Variable declarations of every device in the house
Bulb frontDoorLightOne = "129.67.198.1:12565";
Bulb frontDoorLightTwo = "129.67.198.3:12565";
Window bedRoomWindow = "129.67.198.65:12565";
LawnMower gardenMower = "129.67.198.62:12565";

Thermometer mainThermometer = "129.67.198.65:12565";
MovementSensor frontDoorSensor = "129.67.198.65:12565";
//optional
init(){
    // Code for initial start up of program
}

// Turn on the light if someone is at the door in the evening.
when frontDoorSensor toggle: on{
    if(now > 17:00 && now < 23:00){
        set frontDoorLightOne toggle on;
        set frontDoorLightTwo toggle on;
    }

    wait 60 seconds;

```

```
36      set frontDoorLightOne toggle off;
37      set frontDoorLightTwo toggle off;
38  }
39
40  when 14:00{
41      if(get mainThermometer celsius > 25 ){
42          // Open window completely
43          setBedroomWindowsOpen(100.0);
44      }
45  }
46
47  // Function
48  void setBedroomWindowsOpen(float percentage){
49      set bedRoomWindow openWindow: percentage;
50  }
51
52  // Turn on lawn mower every 2 weeks at 11:00 every summer.
53  every 2 weeks starting 11:00 01d04m2019y{
54      if(01d04m < now && now < 30d09m){
55          set GardenMower toggle on;
56      }
57  }
```

Code snippet 4.2: Pivot example program

In order to support this use case, some types needs to be declared. This happens from line 1-12. First some signal types are declared. These are the signals that the devices can use for either output or input. These signals can have a range between two integers or real numbers, or some ids and values of some predefined signals. For example on = 1 means, that in the code the devices that uses this signal can take the value "on", which is then translated into 1. This is done to improve the readability of Pivot, since *toggle: on;* is assumed to be easier to understand than *toggle: 1;*.

Next the devices are defined. They can then take some signals for either input, output or both. The input means, that the device can receive this kind of signal. Output means, that the device can send a signal of this type to the main server.

Following the type declarations come variable declarations and instantiations. For example two variables of the type Bulb are created. All devices are then set equal to a string. This string can be for example the IP of the device or some other way of communicating to the device.

The init function in line 23 is meant to execute code, that needs to be run before all the following events and functions. In this case it is empty.

All the regular functions and events are located after the init function. The first event executes when the frontDoorSensor outputs a signal of the type "toggle" with the value on. This code is meant to solve subproblem 1. Every event then has a block of code with a local scope. In this block the user can write if-statements, while-statements, assignments and several others. In this case the first event makes

use of the if-statement as well as the "wait". Another important feature shown in this event is the keyword "now". This keyword is resolved at runtime to the current time of the machine when the event is run.

At line 48 a function with the type void starts. The syntax for this is similar to Java or C#. A function can take some input parameters of a specified type as well as return a value of a predefined value. These are meant to generalise some actions and can reduce repeating some code parts.

Another important feature is showcased in the last event starting at line 53. This type of event is called a repeating event. This means that the event executes with a given interval. In this example the event triggers every two weeks at 11:00 starting from the 1st of April in 2019. The format for the date was chosen in order to avoid confusion with American and and European date formats, where the month comes either first or after the first dot respectively. This date format should increase readability for international users. Inside the function an if statement checks if now is in the summer months, and if so sets the garden mower toggle to on.

4.3 Context free Grammar

The grammar of the Pivot language is a context free grammar written in ANTLRs EBNF syntax[1]. It could have been written in normal BNF, but since the ANTLR parsere tool is used, the ANTLR EBNF was chosen too.

In the ANTLR version of EBNF, there is some functionality build in, that can be used in the following ways:
rule* = The rule can be zero or more times
rule? = The rule can be used zero or one time
rule+ = The rule can be use one or more times
rule1 | rule2 = Either rule one or rule two

The first rule simply consists of some declarations and the **EOF** token. This is done, since the Pivot language basically consists of only declarations, that then execute automatically.

The context free grammar can be used to guide the lexical and syntax analysis part of the compiler. The grammar, in this case using ANTLR4, can generate a parse tree if and only if the input program is recognised by the grammar. This is due to the lexer being unable to lex the input word, if no token can be applied, this in turn means, that the parser cannot parse the program.

The grammar, however, cannot handle language features like declaration before use[25], which is also a property of Pivot. This must be handled in the contextual analysis part of the compiler.

The grammar of the Pivot language must also reflect the intended behaviour. For example precedence of operators can be achieved in the grammar. This is also the case for the Pivot language. In the *expr* rule the division and multiplication rules are tested before the addition and subtraction. This means, that the parser tries to derive using the multiplication/addition rule before the addition/subtraction, which puts multiplication further down in the parse tree.

⟨*program*⟩	⇒ ⟨*decls*⟩ EOF	(1)
⟨*decls*⟩	⇒ ⟨*define*⟩* (⟨*declVar*⟩ | ⟨*declDevice*⟩)* ⟨*init*⟩? (⟨*funcDecl*⟩ | ⟨*event*⟩)*	(2)
⟨*define*⟩	⇒ `'define'` (⟨*signal*⟩ | ⟨*device*⟩) `';'`	(3)
⟨*signal*⟩	⇒ `'Signal'` ID `':'` (⟨*range*⟩ | ⟨*enumerations*⟩)	(4)
⟨*enumerations*⟩	⇒ ⟨*enumeration*⟩ (`','` ⟨*enumeration*⟩)*	(5)
⟨*enumeration*⟩	⇒ ID `'='` ⟨*litVal*⟩	(6)
⟨*range*⟩	⇒ ⟨*lowerBound*⟩ `'..'` ⟨*upperBound*⟩	(7)
⟨*lowerBound*⟩	⇒ INTEGER | FLOAT	(8)
⟨*upperBound*⟩	⇒ INTEGER | FLOAT	(10)
⟨*device*⟩	⇒ `'Device'` ID (⟨*inputs*⟩? ⟨*outputs*⟩? | ⟨*outputs*⟩? ⟨*inputs*⟩?)	(12)
⟨*inputs*⟩	⇒ `'input'` `':'` ID (`','` ID)*	(13)
⟨*outputs*⟩	⇒ `'output'` `':'` ID (`','` ID)*	(14)

⟨*declDevice*⟩ ⇒ ID ID '=' STRING ';' (15)

⟨*init*⟩ ⇒ 'init' '(' ')' ⟨*block*⟩ (16)

⟨*funcDecl*⟩ ⇒ (⟨*varType*⟩ | 'void') ID '(' ⟨*fParams*⟩ ')' ⟨*block*⟩ (17)

⟨*fParams*⟩ ⇒ (⟨*param*⟩ (',' ⟨*param*⟩)*)? (18)

⟨*param*⟩ ⇒ ⟨*varType*⟩ ID (19)

⟨*event*⟩ ⇒ ⟨*atomEvent*⟩ (20)
| ⟨*repeatEvent*⟩

⟨*atomEvent*⟩ ⇒ 'when' ID ID ':' (ID | ('exceeds' | 'deceeds') ⟨*number*⟩) ⟨*block*⟩ (22)

⟨*number*⟩ ⇒ INTEGER (23)
| FLOAT

⟨*repeatEvent*⟩ ⇒ 'every' INTEGER ⟨*timeFrame*⟩ ⟨*block*⟩ (25)
| 'every' INTEGER ⟨*timeFrame*⟩ 'starting' ⟨*timeAndDate*⟩ ⟨*block*⟩

⟨*timeAndDate*⟩ ⇒ TIME (DATEnoYEARnoMonth | DATE | DATEnoYEAR)? (27)
| TIME (DATEnoYEARnoMonth | DATEnoYEAR | DATE)

⟨*timeFrame*⟩ ⇒ 'months' (30)
| 'weeks'
| 'days'

			'hours'
			'minutes'
			'seconds'
			'ms'	
⟨*block*⟩	⇒	'{' ⟨*stmts*⟩ '}'	(37)
⟨*stmts*⟩	⇒	(⟨*waitStmt*⟩ | ⟨*assignment*⟩ | ⟨*ifstmt*⟩ | ⟨*whilestmt*⟩ | ⟨*funcCall*⟩ ';' | ⟨*printStmt*⟩ | ⟨*declVar*⟩ | ⟨*rtn*⟩)*	(38)
⟨*printStmt*⟩	⇒	'print' ⟨*expr*⟩ ';'	(39)
⟨*waitStmt*⟩	⇒	'wait' ⟨*expr*⟩ ⟨*timeFrame*⟩ ';'	(40)
⟨*assignment*⟩	⇒	ID '=' ⟨*expr*⟩ ';'	(41)
⟨*ifstmt*⟩	⇒	'if' '(' ⟨*logical_expr*⟩ ')' ⟨*block*⟩ ('else' ⟨*block*⟩)?	(42)
⟨*whilestmt*⟩	⇒	'while' '(' ⟨*logical_expr*⟩ ')' ⟨*block*⟩ ('else' ⟨*block*⟩)?	(43)
⟨*funcCall*⟩	⇒	ID '(' ⟨*arguments*⟩ ')'	(44)
	|	'set' ID ID ':' ⟨*expr*⟩	
	|	'get' ('input' | 'output')? ID ID	
⟨*arguments*⟩	⇒	⟨*expr*⟩? (',' ⟨*expr*⟩)*	(47)
⟨*declVar*⟩	⇒	⟨*varType*⟩ ID '=' ⟨*expr*⟩ ';'	(48)
⟨*rtn*⟩	⇒	'return' ⟨*expr*⟩? ';'	(49)

⟨*litVal*⟩ ⇒ INTEGER (50)
| FLOAT
| STRING

⟨*expr*⟩ ⇒ ⟨*expr*⟩ ('/' | '*') ⟨*expr*⟩ (53)
| ⟨*expr*⟩ ('+' | '-') ⟨*expr*⟩
| '(' ⟨*expr*⟩ ')'
| ⟨*atom*⟩
| ⟨*funcCall*⟩

⟨*logical_expr*⟩ ⇒ ⟨*logical_expr*⟩ '&&' ⟨*logical_expr*⟩ (58)
| ⟨*logical_expr*⟩ '||' ⟨*logical_expr*⟩
| ⟨*comparison_expr*⟩
| '(' ⟨*logical_expr*⟩ ')'

⟨*comparison_expr*⟩ ⇒ ⟨*comparison_operand*⟩ ⟨*comp_operator*⟩ (62)
⟨*comparison_operand*⟩
| '(' ⟨*comparison_expr*⟩ ')'

⟨*comparison_operand*⟩ ⇒ TIME (64)
| (DATE | DATEnoYEAR |
DATEnoYEARnoMonth)
| expr

⟨*comp_operator*⟩ ⇒ '>' (67)
| '>='
| '<'
| '<='
| '=='
| '!='

⟨*atom*⟩ ⇒ ⟨*litVal*⟩ (73)
| ID
| 'now'

⟨*varType*⟩ ⇒ 'string' (76)
| 'int'
| 'float'

4.4 Token specification

The tokens are generated by the lexer and are used by the parser to derive a parse tree. The rules for the lexer is defined in 4.3, where the rules tell the lexer which part of the code should be made into tokens. When the pivot lexer encounters a white space it skips it and jumps to next character. Since indentation is not used to define scopes white space is ignored. The lexer also does not care about newline which enables compound statements in one line. In the pivot language comments are using the syntax **//comment** which is very similar to most general purpose languages when the lexer encounters this it jumps down to next line.

WhiteSpace :	(' '|'\t') -> skip;
NewLine :	NEWLINE -> skip;
LINE_COMMENT :	'//' ~[\r\n]* -> skip;

Table 4.3: Lexer rules

In the ANTLR grammar it is possible to create token fragments, see table 4.4. These are used to build tokens in the grammar, but they cannot be used inside rules. In the case of the Pivot language there are only four fragments.

NEWLINE	::=	('\r' ? '\n' | '\r')
LOWERCASE	::=	[a-z]
UPPERCASE	::=	[A-Z]
DIGIT	::=	[0-9]

Table 4.4: Token Fragments

The first fragment is the NEWLINE. This fragment is used in the lexer rules, see table 4.3. The next fragment consists of lower case characters and is used inside for example variable IDs, since they can contain lower case characters. **[a-z]** in the ANTLR syntax this means all small characters in the ASCII table. This, however, rules out small nation specific characters like for example æ, ø and ä. The same applies to the next fragment consisting of the capital letters. Lastly digits are created, consisting of the numbers zero through nine.

In table 4.5 tokens are defined these consist of either fragments from the table 4.4 or string literals. There are about thirty more tokens, but they are simply string literals consisting of the predefined keywords of the Pivot language, so they were left out.

DATE	::=	DIGIT DIGIT 'd' DIGIT DIGIT 'm' DIGIT DIGIT DIGIT DIGIT 'y'
DATEnoYEAR	::=	DIGIT DIGIT 'd' DIGIT DIGIT 'm'
DATEnoYEARnoMonth	::=	DIGIT DIGIT 'd'
FLOAT	::=	'-'? DIGIT+ '.' DIGIT+
TIME	::=	DIGIT DIGIT ':' DIGIT DIGIT
INTEGER	::=	'-'? DIGIT+
STRING	::=	'"' (LOWERCASE | UPPERCASE | SIGN | DIGIT)* '"'
ID	::=	(LOWERCASE | UPPERCASE) (LOWERCASE | UPPERCASE | DIGIT)*
SIGN	::=	'_'
	|	'-'
	|	'!'
	|	' '
	|	'.'
	|	':'
	|	'+'
	|	'/'
	|	'='
EOF	::=	$

Table 4.5: Tokens

In the table 4.5 it can also be seen, that an ID of a variable must start with a letter, either lower- or uppercase, followed by any length of lower- or uppercase letters or digits. This would would allow for the camelcase convention of naming variables, that is also used in, for example Java[8], which is also the target language.

It is also inside the table of tokens, that the syntax for the types in Pivot is visible. For example the int in the Pivot language can only be lexed, if it follows this syntax, i.e. an optional '-' in front of at least one digit. The string is surrounded by ' " ' on both sides. The float must contain a dot to seperate from numbers to decimals.

Chapter 5

Semantics

Now that the syntax is defined, the meaning of this syntax can be formalised using operational semantics. Furthermore, this chapter includes a description of the type rules system used in the Pivot language.
Operational semantics can be described in two ways: small-step and big-step semantics. The two are fundamentally the same, because they give the same outcome, but at the same time very different in their descriptions of calculations. A big-step transition $\gamma \rightarrow \gamma'$ would describe the whole derivation, starting from a configuration γ, and always go to an end configuration γ'. A small-step transition $\gamma \rightarrow \gamma'$ doesn't have to go to a finished configuration, γ', which means that the small-step semantics describes a part of the derivation. This can be illustrated with the semantics of a while loop. The big-step semantics describes the whole process from the first time the loop runs, to when the while loop terminates, whereas the small-step semantics describes one loop in the while loop. The small-step, for example, can derive a part of a statement, whereas the big-step semantics always derive the entire statement.

The big-step and small-step semantics have their advantages and disadvantages. The big-step semantic is a lot easier to write and also more simple, but in cases of languages that uses any form for parallelism, the big-step semantics cannot be used. The reason for this is, that if two threads run inside the same program, one can get interrupted, while the other thread then runs. This means, that the program will no longer simply run sequentially. One statement may only be half way derived, when the thread is interrupted. For this reason, small-step semantics is the better choice for the Pivot semantics[10].

5.1 Abstract Syntax

After creating some example programs with selected syntax an abstract syntax is developed. The abstract syntax is meant to give the user of the language a better understanding of the structure of a program written in Pivot.

Syntactical categories

P ∈ program
D ∈ Definitions
DC ∈ Declarations
S_D ∈ Signal Definition
D_D ∈ Device Definition
V_D ∈ Variable Declaration
O_D ∈ Object Declaration
E_D ∈ Event Declaration
F_D ∈ Function Declaration
I ∈ Init Function
B ∈ Block
SB ∈ Synchronized Block
F ∈ Function Call
S ∈ Statement

Meta variables:

T ∈ time
t ∈ type
ds ∈ device signals
sl ∈ signal literal
b ∈ boolean
eb ∈ event boolean
v ∈ value
x ∈ var ex ∈ expression

Rules:

P ::= D O_D V_D I DC
D ::= S_D | D_D | D; D | ε

S_D ::= **signal** id: sl | **signal** id: v..v
sl ::= id = v, sl | ε

D_D ::= **device** id **input** s | **device** id **input** s **output** s |
device id **output** s | **device** id **output** s **input** ds |
s ::= id s | ε

V_D ::= t id = ex | V_D;V_D | ε

O_D ::= id id = str | O_D;O_D | ε

I ::= **init()** B

DC ::= F_D | E_D | DC DC | ε
F_D ::= t id (t x) B | ε
E_D ::= eb SB | ε
eb ::= **when** id id v | **when** id **exceeds** v | **when** id **deceeds** v | **every** T
B ::= S; B | ε
SB ::= S; SB | ε
S ::= t x = ex | x = ex | F | **if** b B **else** B | **if** b B | **while** b B | **wait** T
T ::= v **ms** | v **seconds** | v **hours** | v **days** | v **weeks** | v **months**
F ::= id(ex) | **set** id id v | **get** id id
ex ::= v | x | F | ex_1 + ex_2 | ex_1 - ex_2 | ex_1 / ex_2 | ex_1 * ex_2 | (ex)

A program in the Pivot language consists of type declarations, variable declarations, an initialisation function and then a series of event and functions declarations. The words in bold are keywords. The v is used for some value. It can be either a float, an int or some string.

5.2 Small Step Semantics

This section contains all the small step semantic transitions for Pivot. All semantics for the Pivot programming language are of the small-step variant. This is due to the fact, that Pivot programs are likely to be running concurrently and may never terminate. Big-step semantics require, that a statement can be finished atomically in one go. This may, however, not be the case for a Pivot program, since they can be running several threads concurrently. This means, that if two different statements are run simultaneously, a CPU interrupt may stop one statement in favour of another thread from the same program. For this reason some intermediate steps must be formalised. This is done using small-step semantics.

The following semantics in this chapter is based off of Hans Hüttel's book 'Pilen ved træets rod'[10]. Since the book only contains the very base of a programming languages semantics, extensions of the theory from the book had to be made in order to describe the semantics of Pivot.

5.2.1 Definitions

In order to describe the semantics of Pivot, some definitions have to be made first. These definitions consists mostly of functions, that are used in the small step semantics.

$$\textbf{Env V} = \textbf{Var} \cup \{next\} \rightharpoonup \textbf{Loc}$$

The set of variable environments Env V is the partial function from variables to storage locations. The $\{next\}$ represents the next available location in storage.

$$\textbf{Sto} = \textbf{Loc} \rightharpoonup \mathbb{N} \cup \mathbb{R} \cup \Sigma^*$$

The 'Store' describes the values that locations hold. The set of stores is the partial function from locations to the set of all possible values, that is the set of all strings, all natural numbers, and all real numbers.

$$\textbf{Env F} = \textbf{F}_{\textbf{name}} \rightharpoonup \textbf{Block} \times \textbf{Var} \times \textbf{EnvV} \times \textbf{EnvF}$$

Both variables and functions are statically scoped, therefore a function environment must remember the variable and function environments that existed when the function was declared. Additionally it must remember any input parameters, here denoted 'Vars'. For these reasons, the set of function environments is defined recursively as the partial function from function names to the Cartesian product of blocks (containing statements), input parameters, variable environments, and function environments.

$$\textbf{Env E} = \textbf{E}_{\textbf{signature}} \rightharpoonup \textbf{SyncBlock} \times \textbf{EnvV} \times \textbf{EnvF}$$

Similarly, due to static scoping, an event environment must also remember the variable bindings and function environment that existed at the time of the event declaration. The set of event environments is defined as a partial function from event signatures to the cartesian product of statements, variable environments and function environments.

$$\textbf{Glo} = \textbf{Stmnt} \rightharpoonup \mathcal{P}(\textbf{Var})$$

The glo function maps a statement to the global variables used within that statement. As an example $glo[\![x = a + b]\!]$ will yield {b, x} if the variables x and b are declared in the global scope, and a is declared locally.

$$\mathbf{Lock} \;=\; \mathbf{Var} \rightharpoonup (\mathbf{Stmnt}) \cup \mathit{ff}$$

For the sake of safely executing statements in parallel, a system with global variable locks is used. If multiple statements use the same global variable, only the statement currently holding the lock on that global variable, will be able to execute. Only one statement can hold the lock on a specific global variable at any point in time. This behaviour is explained and specified further in the transition rules for statements. The set of all locks is the partial function from variables to statements and the value false (*ff*). The value of *lock x* is the statement currently holding the lock. This value is false if no statement currently holds the lock.

$$\mathbf{EnvS} \;=\; \mathbf{name} \rightharpoonup \mathbf{Env_{vs}}$$

To describe the signal type definition, a signal environment Env S is introduced. This environment is a function from a signal name to a variable environment inside that signal. This variable environment includes all variables inside the signal. This includes for example the current and previous value of that variable.

$$\mathbf{EnvD} \;=\; \mathbf{name} \rightharpoonup \mathbf{Env_{sout}} \times \mathbf{Env_{sin}}$$

Similarly a device environment is introduced for describing device type definitions. This is a partial function, that takes a device type name as argument and then returns two signal environments consisting of signals of either in- or output.

$$\mathbf{EnvO} \;=\; \mathbf{name} \rightharpoonup \mathbf{Env_{sout}} \times \mathbf{Env_{sin}}$$

Lastly the Env_O is defined. The O stands for object and describes device object declarations. This is a partial function from a device name to a new signal environment for both out- and input signals respectively.

5.2.2 Boolean and Arithmetic Semantics

While developing the Pivot language, there were multiple needs for various boolean expressions. These were needed in order to compare signals from devices and variables. The functionality of the boolean expressions used in pivot, are equivalent to the same mathematical expressions. Though in pivot, other symbols are used to describe the same formal mathematics, where e.g. logical **or** would be described in mathematical notation as $\vee$, and as $||$ in pivot. For the concrete boolean and arithmetic semantics, see section B in the appendix. The semantics for the special event-boolean expressions **exceeds** and **deceeds**, which are mentioned in section 4.1.5, are explained in section B.1.

5.2.3 Small-step semantics for values

The semantics for declaring variables and the predefined keyword *now* is described in this section. Variables can be declared and assigned values, which is described using the environment store model. In the variable environment, x has the address of l, and in the store, the address l has the value of v($env_v\, x = l$). The *now* keyword is essential for the pivot language since events should be able to be timebased. When the now keyword is used, the *currentTime* on the machine is fetched in runtime, making it possible to create time intervals, where something is executed. E.g. using if statements: $if(now > 17:00 \;\&\&\; now < 21:00)$

[VAR]

$$env_v, sto \vdash x \rightarrow_e v \qquad \text{if } env_v\, x = l \text{ and } sto\, l = v$$

[NOW]

$$sto \vdash now \Rightarrow v$$
where $v = currentTime$ and $sto\, currentTime = v$

Table 5.1: Small-step semantics for values

5.2.4 Synchronized Blocks

Pivot supports running events, i.e. statements, in parallel, the semantic specification for this is detailed in section 5.2.14. This can cause concurrency problems as multiple events could be manipulating the same global variables at the same time. To accommodate this problem, synchronized blocks are introduced.

Synchronized blocks provide an implicit, form of thread safety when programming in Pivot. As seen in the abstract syntax in section 5.1, events have synchronised blocks defining the behaviour of the events. A statement contained within a synchronised block must acquire locks on all global variables it contains, before the statement can be evaluated. Locks are global, and a variable may only be locked by one statement at a time. This ensures that statements running in parallel will not interfere with each other by manipulating the same variables at the same time. The use of locks can potentially cause a deadlock where concurrent statements are stuck in an infinite cycle waiting for each other. Deadlocks of this type can be avoided by always acquiring locks in the same order. Therefore, variable locks in pivot are always acquired in order of when they were defined[19].

[SYNC-BLOCK-1]

$$env_v, env_f \vdash < S_1; SB,\ sto,\ lock > \Rightarrow < S_1; SB,\ sto,\ lock[v \mapsto (S_1)] >$$
$$\textbf{if } nonlocked \neq \varnothing \ \ and\ lock\ v = ff$$
$$\textbf{where } v = min\ (nonlocked)$$
$$nonlocked = \{x\ \in\ glo[\![S_1]\!] \mid lock\ x \neq (S_1)\}$$

If a statement in a synchronised block does not hold locks for all global variables used within that statement, it must first acquire locks for those variables one at a time. The set *nonlocked* is the set of all the global variables contained within S_1, that are not currently locked to S_1. Since all locks must be acquired in the same order, the global variable that was defined first (here denoted *v=min(nonlocked)* will be locked. The lock can only be acquired if no other statement currently holds the lock, therefore it is a requirement that $lock\ v = false$. In the **[]SYNC-BLOCK-1]** transition, the statement S_1 acquires the lock on the variable v, by mapping *lock v* to the statement S_1.

[SYNC-BLOCK-2]

$$\frac{env_v, env_f \vdash < S_1, sto > \Rightarrow < S_1',\ sto' >}{env_v, env_f \vdash < S_1\ ;\ SB, sto,\ lock > \Rightarrow < S_1';\ SB, sto',\ lock >}$$
$$\textbf{if nonlocked} = \varnothing$$
$$\textbf{Where } nonlocked = \{x\ \in\ glo[\![S_1]\!] \mid lock\ x \neq (S_1)\}$$

A statement may be executed, or partially executed, when it holds the lock on all global variables contained within the statement.

[SYNC-BLOCK-3]

$$\frac{env_v, env_f \vdash < S_1, sto > \Rightarrow sto'}{env_v, env_f \vdash < S_1\ ;\ SB, sto, lock > \Rightarrow < SB,\ sto',\ lock' >}$$
$$\textbf{if nonlocked} = \varnothing$$
$$\textbf{Where } lock' = lock[v \mapsto ff]\ for\ all\ v\ \in glo[\![S_1]\!]$$
$$nonlocked = \{x\ \in\ glo[\![S_1]\!] \mid lock\ x \neq (S_1)\}$$

Once a statement finishes executing it releases all the locks it held, by mapping the variables to false in the lock function.

5.2.5 Signal Type Definitions

[SIGNAL-ENUM$_{\text{DEF-1}}$]

$$\frac{< sl >\Rightarrow< x = v; sl' >}{< signal\ id_s : sl, env_s, sto >\Rightarrow< signal\ id_s : sl, env_s[id_s \mapsto\ env'_v], sto' >}$$

$$\begin{gathered}\text{Where } env'_v = env_v[x \mapsto l][next \mapsto new\ l] \\ env_v = env_s\ id_s \\ sto[l \mapsto v][] \\ l = env_v\ next\end{gathered}$$

[SIGNAL-ENUM$_{\text{DEF-2}}$]

$$\frac{< sl >\Rightarrow< x = v >}{< signal\ id_s : sl, env_s, sto >\Rightarrow< env_s[id_s \mapsto\ env'_v], sto' >}$$

$$\begin{gathered}\text{Where } env'_v = env_v[x \mapsto l][next \mapsto new\ l] \\ env_v = env_s\ id_s \\ sto[l \mapsto v][] \\ l = env_v\ next\end{gathered}$$

[SIGNAL-RANGE$_{\text{DEF}}$]

$$< Signal\ id_s : v_1..v_2;, env_s, sto >\Rightarrow< env_s[id_s \mapsto env'_v], sto' >$$

$$\begin{gathered}\text{Where } env'_v = env_v[lowBound \mapsto l_1][upBound \mapsto l_2] \\ [currentVal \mapsto l_3][prevVal \mapsto l_4][next \mapsto newl] \\ env_v = env_s\ id_s \\ sto = sto'[l_1 \mapsto v_1][l_2 \mapsto v_2][l_3 \mapsto v_1]\end{gathered}$$

The **[SIGNAL-RANGE]** defines a signal based on a range of values. In order to describe a defined range, we use the boundaries *lowBound* and *upBound*. The default value is set to be the *lowBound*, and is described in the *store* condiditon, where $[l_3 \mapsto v_1]$

5.2.6 Device Type Definitions

Device type definitions create new definitions in the device environment env_d. This is done by mapping the device type name to the given input and output signals in the device environment.

[DEVICE-DEF-1]

$$\frac{< s_i >\Rightarrow< id_{\text{signal}};\ s_i';>}{env_s \vdash< \texttt{Device}\ x\ \texttt{input:}\ s_i\ \texttt{output:}\ s_o, env_d >\Rightarrow \\ < \texttt{Device}\ x\ \texttt{input:}\ s_i'\ \texttt{output:}\ s_o,\ env_d[x \mapsto (env_{\text{s\_in}}',\ env_{\text{s\_out}})] >}$$

$$\textbf{where}\ env_d\ x = (env_{\text{s\_in}}, env_{\text{s\_out}}) \\ env_{\text{s\_in}}' = env_{\text{s\_in}}[id_{\text{signal}} \mapsto signal] \\ signal = env_s\ id_{\text{signal}}$$

In rule [DEVICE-DEF-1] input signals are added to the device environment one at a time. Here the input signal evaluates to a signal id followed by more input signals. The first signal is added by looking up the signal id in the signal environment, and then mapping it to device x in the device environment.

[DEVICE-DEF-2]

$$\frac{< s_i >\Rightarrow< id_{\text{signal}};>}{env_s \vdash< \texttt{Device}\ x\ \texttt{input:}\ s_i\ \texttt{output:}\ s_o, env_d >\Rightarrow \\ < \texttt{Device}\ x\ \texttt{output:}\ s_o,\ env_d[x \mapsto (env_{\text{s\_in}}',\ env_{\text{s\_out}})] >}$$

$$\textbf{where}\ env_d\ x = (env_{\text{s\_in}}, env_{\text{s\_out}}) \\ env_{\text{s\_in}}' = env_{\text{s\_in}}[id_{\text{signal}} \mapsto signal] \\ signal = env_s\ id_{\text{signal}}$$

In rule [DEVICE-DEF-2] the input signal evaluates to just one signal id. The signal is added to the device in the device environment, and the whole definition evaluates to a device definition with only the output signals.

[DEVICE-DEF-3]

$$\frac{< s_o > \Rightarrow < id_{\text{signal}}; s_o' >}{env_s \vdash < \texttt{Device } x \texttt{ output: } s_o, env_d > \Rightarrow \\ < \texttt{Device } x \texttt{ output: } s_o',\ env_d[x \mapsto (env_{\text{s\_in}},\ env_{\text{s\_out}}')] >}$$

$$\textbf{where } env_d\ x = (env_{\text{s\_in}}, env_{\text{s\_out}}) \\ env_{\text{s\_out}}' = env_{\text{s\_in}}[id_{\text{signal}} \mapsto signal] \\ signal = env_s\ id_{\text{signal}}$$

[DEVICE-DEF-4]

$$\frac{< s_o > \Rightarrow < id_{\text{signal}} >}{env_s \vdash < \texttt{Device } x \texttt{ output: } s_o, env_d > \Rightarrow \\ env_d[x \mapsto (env_{\text{s\_in}},\ env_{\text{s\_out}}')]}$$

$$\textbf{where } env_d\ x = (env_{\text{s\_in}}, env_{\text{s\_out}}) \\ env_{\text{s\_out}}' = env_{\text{s\_in}}[id_{\text{signal}} \mapsto signal] \\ signal = env_s\ id_{\text{signal}}$$

5.2.7 Device declaration

[DEVICE-DECL]

$$< id_{\text{type}}\ \ id_{\text{object}} = str,\ env_o r > \Rightarrow env_o[id_{\text{type}} \mapsto (env_d\ id_{\text{type}})]$$

[DEVICE-DECL]

$$< \epsilon,\ env_o r > \Rightarrow env_o$$

Table 5.9: Defines and device declaration

5.2.8 Variable Declaration

The variable declaration can either partly evaluate the expression as in [VAR-DECL-1] or completely, like in [VAR-DECL-2]. The first one only changes the expression. Both needs the function environment to look up any function calls,

that might be part of the expression. When the expression is completely evaluated, it is now a value. Now all locks will be released and the value is stored at location l. For this reason the variable environment, the store and the lock is changed.

In the variable declarations the new function is used to find a new l, which *next* is then bound to. This simply means, that the *next* is bound to a new location in memory. This is used to give a new variable some new location is memory.

The new function therefore looks like this:

$$new : loc \mapsto loc$$

This function maps from one location to a new and empty one.

[VAR-DECL-1]
$$\frac{env_f, env_v, sto \vdash < ex > \Rightarrow < ex' >}{env_f \vdash < t\ x = ex;\ V_D,\ env_v,\ sto,\ lock > \Rightarrow \text{<t x = ex'; V\_D, env\_v, sto, lock>}}$$

[VAR-DECL-2]
$$\frac{env_v, env_f, sto \vdash < ex > \Rightarrow v}{env_f \vdash < t\ x = ex;\ V_D,\ env_v,\ sto,\ lock > \Rightarrow \text{<V\_D, env\_v', sto', lock'>}}$$

Where

$$env'_v = env_v[x \mapsto l][\{next\} \mapsto new\ l]$$
$$l = env_v\ next$$
$$sto' = sto[l \mapsto v]$$
$$lock' = lock[x \mapsto ff]$$

5.2.9 Function Declaration

The function declaration can change the function environment by adding the function declared.

[FUNC-DECL-1] $$env_v \vdash < t\ f\ (t\ x)\ B,\ env_f > \Rightarrow\ env_f[f \mapsto (B, x, env_v, env_f)]$$

[FUNC-DECL-2] $$env_v,\ env_f \vdash < \varepsilon,\ env_f > \Rightarrow env_f$$

5.2.10 Event Declaration

An event declaration changes the event environment so that, the new event is added. An event contains a synchronized block (SB), an event boolean (eb), a vari-

able environment as well as a function environment. The empty event declaration does nothing.

[EVENT-DECL-1]
$$env_v,\ env_f\ \vdash < eb :\ SB,\ env_e > \Rightarrow \\ env_e[(eb : SB) \mapsto\ (SB,\ eb,\ env_v,\ env_f)]$$

[EVENT-DECL-2]
$$env_v,\ env_f\ \vdash < \varepsilon,\ env_e > \Rightarrow env_e$$

5.2.11 Statement Transitions

The statements transitions such as assignments, if statements, function calls and so on are described in this section.

[ASSIGN-1]
$$\frac{env_v,\ env_f\ \vdash < ex, sto > \Rightarrow < ex', sto' >}{env_v, env_f \vdash < e\ =\ ex, sto > \Rightarrow < x\ =\ ex', sto' >}$$

[ASSIGN-2]
$$\frac{env_v, env_f \vdash < ex > \Rightarrow v}{env_v, env_f \vdash < x\ =\ ex, sto > \Rightarrow\ sto[x\ \mapsto\ v]}$$

If the ex has not yet evaluated to a val, like in [ASSIGN-1] the ex is further evaluated. This partly evaluation may, however, also change the store, since an expression can call a function with side effects. Otherwise the assignment statement changes the value of location x to value v.

[PRINT-1]
$$\frac{env_v,\ env_f\ \vdash < ex, sto > \Rightarrow < ex', sto' >}{env_v, env_f\ \vdash < \texttt{print}\ ex,\ sto' > \Rightarrow < \texttt{print}\ ex', sto' >}$$

[PRINT-2]
$$\frac{env_v,\ env_f\ \vdash < ex, sto > \Rightarrow (v, sto')}{env_v, env_f\ \vdash < \texttt{print}\ ex,\ sto > \Rightarrow < \texttt{print}\ ex, sto'[l \rightarrow v] >}$$

The if statements have several outcomes depending on the value of the boolean. If the boolean is true, the if statement simply evaluates to the block contained within the if statement. If the boolean is false the if statement simply evaluates to no change in the store.

[IF-TRUE]
$$env_v, env_f\ \vdash < \texttt{if}\ b\ B,\ sto > \Rightarrow < B, sto > \\ \textbf{if}\ env_v,\ env_f \vdash b \rightarrow tt$$

[IF-FALSE] $$env_v, env_f \vdash < \texttt{if}\ b\ B,\ sto > \Rightarrow sto \quad \textbf{if}\ env_v, env_f \vdash b \rightarrow ff$$

Each if statement can also contain a single else clause. The else clause also contains a block. If the if statement boolean is false, the statement evaluates to the else block, otherwise the other way around.

[IF-TRUE-ELSE] $$env_v, env_f \vdash < \texttt{if}\ b\ B_1\ \texttt{else}\ B_2,\ sto > \Rightarrow < B_1, sto > \quad \textbf{if}\ env_v,\ env_f \vdash b \rightarrow tt$$

[IF-FALSE-ELSE] $$env_v, env_f \vdash < \texttt{if}\ b\ B_1\ \texttt{else}\ B_2,\ sto > \Rightarrow < B_2,\ sto > \quad \textbf{if}\ env_v, env_f \vdash b \rightarrow ff$$

The while statement uses the if statement to derive the next step. This can be done, since *if b (B; while b B))* is equivalent in semantic meaning to *while b B*. Then the if statement derivation is simply used to continue.

[WHILE] $$env_v, env_f < \texttt{while}\ b\ \texttt{B, sto} > \Rightarrow < \texttt{if}\ b\ \texttt{(B; while b B), sto} >$$

The wait statement in the Pivot language does not change anything in the store. It simply checks if the time has passed, then the execution is continued with the same store.

[WAIT-DONE]

$$env_v, env_f < \texttt{wait}\ ex\ \texttt{T}, sto >\Rightarrow sto$$
$$\text{If } now = startTime + v\ T$$
$$\text{Where } env_v, sto \vdash ex \rightarrow_{\text{ex}} v$$

[WAIT-NOT-DONE]

$$env_v, env_f < \texttt{wait}\ ex\ \texttt{T}, sto >$$

$$\text{If } now \neq startTime + v\ T$$
$$\text{Where } env_v, sto \vdash ex \rightarrow_{\text{ex}} v$$

[FUNCTION$_{\text{CALL}}$]

$$\frac{\begin{array}{c} env'_v[x_1 \mapsto l_1][x_2 \mapsto l_2]..[x_n \mapsto l_n][next \mapsto new\, l_n], \\ env'_f[p \mapsto (\texttt{S}, x_1, x_2, ..\, x_n, env'_v, env'_f)] \\ \vdash < \texttt{S}, sto[l_1 \mapsto v_1][l_2 \mapsto v_2]\,..\,[l_n \mapsto v_n] >\rightarrow sto' \end{array}}{env_v, env_p \vdash < p(ex_1, ex_2, ..\, ex_n), sto >\rightarrow sto'}$$

$$\text{Where } env_f\ p = (\texttt{S}, x_1, x_2, ..\, x_n, env'_v, env'_f)$$
$$\text{Where } env_v, sto \vdash ex_1 \rightarrow_e xv_1, sto \vdash ex_2 \rightarrow_e xv_2, ..\, sto \vdash ex_n \rightarrow_e xv_n$$
$$\text{Where } l = env_v\, next$$

[SET-FUNC]

$$\frac{env_f, env_v, sto \vdash < ex >\Rightarrow v}{env_v, env_f\ \vdash < \texttt{set}\ \ dev\ sig\ :\ ex, sto, env_o >\Rightarrow (sto, env_o[dev \mapsto env'_s])}$$

$$\text{Where } env_o\ dev = env_s$$
$$env'_s\ = env_s[sig \mapsto env'_{vs}]$$
$$env_s\ sig\ = env_{vs}$$
$$env'_{vs}\ = env_{vs}[current \mapsto v][prev \mapsto prev]$$
$$prev = env_v\ currentValue$$

[GET-FUNC$_{\text{INPUT}}$]

$$env_v, env_o \vdash < get\ input\ dev\ sig > \Rightarrow < v >$$

Where $env_o\ dev = (env_{\text{s\_in}}, env_{\text{s\_out}})$
$env_{\text{s\_in}}\ sig = env_{\text{vs}}$
$v = env_{vs} current$

[GET-FUNC$_{\text{OUTPUT}}$]

$$env_v, env_o \vdash < get\ output\ dev\ sig > \Rightarrow < v >$$

Where $env_o\ dev = (env_{\text{s\_in}}, env_{\text{s\_out}})$
$env_{\text{s\_out}}\ sig = env_{\text{vs}}$
$v = env_{vs} current$

[GET-FUNC$_{\text{EMPTY}}$]

$$env_v, env_o \vdash < get\ dev\ sig > \Rightarrow < get\ output\ dev\ sig >$$

Table 5.18: Semantics for statement operations

5.2.12 Event Booleans

Event booleans describes the conditions that need be true for an event to be triggered.

Exceeds/deceeds (boolean)

The keywords *exceeds* and *deceeds* are used for a special feature in Pivot. For example if a sensor outputs a value higher than the previous output, it is considered to have exceeded, and the event boolean will evaluate to true. A similar behaviour, but the other way around, is created with the *deceeds* keyword. The rules for *exceeds* can be seen in table 5.19 and the rules for *deceeds* can be seen in table 5.20.

[EXCEEDS-STAYBELOW]

$$
\begin{gathered}
env_O \vdash < \texttt{when}\ \texttt{ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{exceeds}\ \ \texttt{v}_1 > \Rightarrow_{eb} ff \\
\text{If } v_1 > v_2 \\
v_1 > v_3 \\
\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin}) \\
env_{sout} ID_{signal} = env_{vs} \\
env_{vs}\ \texttt{current} = v_2 \\
env_{vs}\ \texttt{prev} = v_3
\end{gathered}
$$

[EXCEEDS-DECEED]

$$
\begin{gathered}
env_O \vdash < \texttt{when}\ \texttt{ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{exceeds}\ \ \texttt{v}_1 > \Rightarrow_{eb} ff \\
\text{If } v_1 > v_2 \\
v_1 < v_3 \\
\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin}) \\
env_{sout} ID_{signal} = env_{vs} \\
env_{vs}\ \texttt{current} = v_2 \\
env_{vs}\ \texttt{prev} = v_3
\end{gathered}
$$

[EXCEEDS-STAYABOVE]

$$
\begin{gathered}
env_O \vdash < \texttt{when}\ \texttt{ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{exceeds}\ \ \texttt{v}_1 > \Rightarrow_{eb} ff \\
\text{If } v_1 > v_2 \\
v_1 > v_3 \\
\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin}) \\
env_{sout} ID_{signal} = env_{vs} \\
env_{vs}\ \texttt{current} = v_2 \\
env_{vs}\ \texttt{prev} = v_3
\end{gathered}
$$

[EXCEEDS-TRUE]

$$env_O \vdash < \texttt{when}\ \texttt{ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{exceeds}\ \ \texttt{v}_1 > \Rightarrow_{eb} tt$$
$$\text{If } v_1 < v_2$$
$$v_1 > v_3$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_{vs}\ \texttt{current} = v_2$$
$$env_{vs}\ \texttt{prev} = v_3$$

Table 5.19: Exceeds semantics

[DECEED-STAYBELOW]

$$env_O \vdash < \texttt{when}\ \texttt{ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{deceeds}\ \ \texttt{v}_1 > \Rightarrow_{eb} ff$$
$$\text{If } v_1 > v_2$$
$$v_1 > v_3$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_{vs}\ \texttt{current} = v_2$$
$$env_{vs}\ \texttt{prev} = v_3$$

[DECEED-STAYBELOW]

$$env_O \vdash < \texttt{when}\ \texttt{ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{deceeds}\ \ \texttt{v}_1 > \Rightarrow_{eb} ff$$
$$\text{If } v_1 < v_2$$
$$v_1 > v_3$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_{vs}\ \texttt{current} = v_2$$
$$env_{vs}\ \texttt{prev} = v_3$$

[DECEED-STAYABOVE]

$$env_O \vdash < \texttt{when ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \texttt{deceeds v}_1 > \Rightarrow_{eb} ff$$
$$\text{If } v_1 < v_2$$
$$v_1 < v_3$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_{vs}\ \texttt{current} = v_2$$
$$env_{vs}\ \texttt{prev} = v_3$$

[DECEED-TRUE]

$$env_O \vdash < \texttt{when ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \texttt{deceeds v}_1 > \Rightarrow_{eb} tt$$
$$\text{If } v_1 > v_2$$
$$v_1 < v_3$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_{vs}\ \texttt{current} = v_2$$
$$env_{vs}\ \texttt{prev} = v_3$$

Table 5.20: Deceeds semantics

When-Event Booleans

The *when* event booleans are used for creating both time based and signal based events.

[WHEN-TIMEANDDATE_TRUE]

$$env_D \vdash < \texttt{when timeAndDate} > \Rightarrow tt$$
$$\text{If } (\texttt{now} \geq timeAndDate)$$
$$\text{where } timeAndDate = env_V timeAndDate$$

[WHEN-TIMEANDDATE_FALSE]

$$env_D \vdash < \texttt{when timeAndDate} > \Rightarrow ff$$
$$\text{If } (\texttt{now} < timeAndDate)$$
$$\text{where } timeAndDate = env_V timeAndDate$$

[WHEN-SIGNAL$_{\text{TRUE}}$]

$$env_D \vdash < \texttt{when ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{v}_1 > \Rightarrow_{eb} tt$$
$$\text{If } v_1 = v_2$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_v\ current = v_2$$

[WHEN-SIGNAL$_{\text{FALSE}}$]

$$env_D \vdash < \texttt{when ID}_{\text{device}}\ \texttt{ID}_{\text{signal}}\ \ \texttt{v}_1 > \Rightarrow_{eb} ff$$
$$\text{If } v_1 \neq v_2$$
$$\text{Where } env_O \texttt{ID}_{device} = (env_{sout}, env_{sin})$$
$$env_{sout} ID_{signal} = env_{vs}$$
$$env_v\ current = v_2$$

Table 5.21: When event semantic

[EVERY-TIMEFRAME_TRUE]

$$env_D \vdash < \texttt{every timeFrame}, env_V > \Rightarrow$$
$$< tt,\ \texttt{env\_V[execTime = now + timeFrame]}$$
$$\text{If } (\texttt{now == } execTime)$$
$$\text{where } execTime = env_V\ \texttt{execTime}$$

[EVERY-TIMEFRAME_FALSE]

$$env_D \vdash < \texttt{every timeFrame}, env_V > \Rightarrow < \texttt{ff} >$$
$$\text{If } (\texttt{now } \neq \texttt{ execTime})$$
$$\text{where } execTime = env_V\ \texttt{execTime}$$

Table 5.22: Every event semantic

5.2.13 Init

[INIT] $< \texttt{init () S}, s > \Rightarrow < \texttt{S}, s >$

5.2.14 Parallelism and Implicit Semantics

Pivot is an event driven language and therefore some actions within the program are not syntax driven, but instead triggered implicitly. To describe these actions, three new semantic concepts are introduced: event execution (denoted E), update and event trigger.

[UPDATE]
$$env_e \vdash < update,\ sto > \Rightarrow < trigger(events),\ sto' >$$
$$\textbf{where } sto' = sto[val \mapsto updatedVal]$$
$$events = e \in env_e$$

[UPDATE]
$$env_e \vdash < update,\ sto > \Rightarrow < trigger(events),\ sto' >$$
$$\textbf{where } sto' = sto[val \mapsto updatedVal]$$
$$events = e \in env_e$$

An **update** is issued whenever an outside event interacts with the program, such as a signal being received or a certain time frame passing. When an external change triggers an update, the **store** is updated to reflect the changes that were made.

[TRIGGER-1]

$$env_e,\ env_v \vdash < trigger(E_1,\ E_2...E_n) > \Rightarrow < \{|trigger(E_2...E_n),\ E_1|\} >$$
$$\textbf{if } env_v,\ sto \vdash eb \Rightarrow tt$$
$$\textbf{where } env_e\ E_1 = (SB_1,\ eb_1,\ env'_v,\ env'_f)$$

[TRIGGER-2]

$$env_e,\ env_v \vdash < trigger(E_1,\ E_2...E_n) > \Rightarrow < trigger(E_2...E_n) >$$
$$\textbf{if } env_v,\ sto \vdash eb \Rightarrow ff$$
$$\textbf{where } env_e\ E_1 = (SB_1,\ eb_1,\ env'_v,\ env'_f)$$

The notation $\{|S_1, S_2, ...|\}$ denotes that all statement within the brackets can be executed at the same time in parallel. The **trigger** start the event execution in parallel if its event boolean *eb* evaluates to true, otherwise the event is ignored.

[EVENT-EXEC-1]

$$\frac{env'_v, env'_f \vdash < SB_1, sto, lock > \Rightarrow < SB'_1,\ sto', lock' >}{env_v,\ env_e \vdash < E_1, sto, lock > \Rightarrow < SB'_1, lock', sto' >} \quad \textbf{Where } env_e\ E_1 = (SB_1, eb_1, env'_v, env'_f)$$

[EVENT-EXEC-2]

$$\frac{env'_v, env'_f \vdash < S_1, sto > \Rightarrow < sto' >}{env_v,\ env_e \vdash < E_1, sto > \Rightarrow < sto' >} \quad \textbf{Where } env_e\ E_1 = (SB_1, eb_1, evn'_v, env'_f)$$

5.3 Type rules

The purpose of the type rules is to show that not all types are compatible and that not all variables can have the same types in the Pivot language. Furthermore, the purpose is to show and formalise the compatibility between different types and different semantic rules. Firstly the type environment and the Pivot environment update must be described. Secondly it is described what the type rules do, and lastly it is described how the type rules are read.
The type environment is described by:

$$K : \textbf{Var} \cup \textbf{Fnames} \cup \textbf{Dnames} \cup \textbf{Snames} \cup \textbf{Date} \rightharpoonup \textbf{Types}$$

Where:
Var: All variables
Fnames: All function names
Dnames: All device names
Snames: All signal names
Date: All dates

To update the type environment the following notation is used:

$$K[x \mapsto C]$$

Which is described by:

$$K'(y) = \begin{cases} K(y) & \text{if } y \neq x \\ C & \text{if } y = x \end{cases}$$

Where C is all types.

We define the rules by $K \vdash k : C$ this means that k has the type C in the type environment K.

5.3.1 Types

There are several different types in the Pivot language, see table 5.28. There are some primitive types e.g. bool, int, float and string, representing truth values, integers, decimal point numbers and text strings respectively. Additionally, some new types e.g. Signal and Device were introduced. The type Date is different from the others. This type describes the variable as a timestamp. This timestamp can either be a time, date or both.

V	::=	Void
D	::=	Date
SD	::=	Signal | Device
LIT	::=	Int | Float | String
B	::=	Bool | Int | Float
A	::=	B | String | x : C $\rightarrow$ ok
C	::=	A | LIT | SD | D | V

Table 5.28: Types

Two variables can be two different types, and therefore they are not compatible in some semantical rules. For example, the type rule ADD in table 5.31 and the VAR_{EXP} in table 5.29 can make out the expression:

$$x = x + y$$

If x is of type Int and y is of type Bool, then this expression cannot be computed in the Pivot language because of the ADD rule. If x is of type Float and y is of type Int, then it is still not compatible in the Pivot language language because of the ADD rule. All elements of the rule must be of the same type in order for the expression to be type correct.

To describe one or more arguments in the type rules a vector is used, as seen in rule FUNC_{DEC} in table 5.32. With the vector it can be shown that a function declaration can have several arguments, shown with $\vec{x}$, of some type, shown with $\vec{A}$. This means that not all the arguments need to have the same type, they just need to be in the A type category.

First in figure 5.3.2 the basic expressions are described.

5.3.2 Type rules

$$[\text{NUM}_{\text{EXP}}] \qquad \frac{K(v) = LIT}{K \vdash v : LIT}$$

[VAR_{EXP}]	$\dfrac{K(x) = A}{K \vdash x : A}$
[$\text{PARENT}_{\text{EXP}}$]	$\dfrac{K \vdash k_1 : A}{K \vdash (k_1) : A}$
[NOW_{EXP}]	$K \vdash \texttt{now} : D$

Table 5.29: Expression typerules

In figure 5.30 the boolean types are described. For example the [EQUALS] boolean statement is only type correct if both sides of the [EQUALS] are of the type boolean.

[EQUALS]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{K \vdash k_1 == k_2 : Bool}$
[SMALLER-THAN]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{K \vdash k_1 < k_2 : Bool}$
[BIGGER-THAN]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{K \vdash k_1 > k_2 : Bool}$
[NOT-EQUAL]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{K \vdash k_1 \texttt{!} = k_2 : Bool}$
[GREATER-OR-EQUAL]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{K \vdash k_1 <= k_2 : Bool}$
[SMALLER-OR-EQUAL]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{E \vdash k_1 >= k_2 : Bool}$
[AND]	$\dfrac{K \vdash k_1 : Bool \quad K \vdash k_2 : Bool}{K \vdash k_1 \&\& k_2 : Bool}$

$$[\text{OR}] \qquad \frac{K \vdash k_1 : Bool \qquad K \vdash k_2 : Bool}{K \vdash k_1 \,||\, k_2 : Bool}$$

Table 5.30: Boolean typerules

In figure 5.31 the arithmetic expression types rules are defined. Here it is visible, that no implicit type conversion is possible. All arithmetic expressions require both sides to be of the same type.

$$[\text{ADD}] \qquad \frac{K \vdash k_1 : LIT \qquad K \vdash k_2 : LIT}{K \vdash k_1 + k_2 : LIT}$$

$$[\text{SUB}_{\text{INT}}] \qquad \frac{K \vdash k_1 : Int \qquad K \vdash k_2 : Int}{K \vdash k_1 - k_2 : Int}$$

$$[\text{SUB}_{\text{FLOAT}}] \qquad \frac{K \vdash k_1 : Float \qquad K \vdash k_2 : Float}{K \vdash k_1 - k_2 : Float}$$

$$[\text{MULT}_{\text{INT}}] \qquad \frac{K \vdash k_1 : Int \qquad K \vdash k_2 : Int}{K \vdash k_1 * k_2 : Int}$$

$$[\text{MULT}_{\text{FLOAT}}] \qquad \frac{K \vdash k_1 : Float \qquad K \vdash k_2 : Float}{K \vdash k_1 * k_2 : Float}$$

$$[\text{DIV}_{\text{INT}}] \qquad \frac{K \vdash k_1 : Int \qquad K \vdash k_2 : Int}{K \vdash k_1 / k_2 : Int}$$

$$[\text{DIV}_{\text{FLOAT}}] \qquad \frac{K \vdash k_1 : Float \qquad K \vdash k_2 : Float}{K \vdash k_1 / k_2 : Float}$$

Table 5.31: Arethmetic type rules

In figure 5.32 the declaration type rules are defined.

$$[\text{EMPTY}_{\text{DEC}}] \qquad E \vdash \varepsilon : ok$$

[VAR$_{\text{DEC}}$]
$$\frac{K[x \mapsto A] \vdash V : ok \qquad K \vdash ex : A}{K \vdash A\ x{=}ex; V : ok}$$

[FUNC$_{\text{DEC}}$]
$$\frac{\begin{array}{c}K \vdash S : ok \quad K \vdash A_1 : T \quad K[p \mapsto A_1] \vdash V : ok \\ K[p \mapsto (\vec{x} : \vec{A} \to ok)] \vdash F : ok\end{array}}{K \vdash A_1\ p(\vec{A}\ \vec{x})\ S; V, F : ok}$$

[FUNC$_{\text{DEC-VOID}}$]
$$\frac{\begin{array}{c}K \vdash S : ok \quad K[p \mapsto Void] \vdash V : ok \\ K[p \mapsto (\vec{x} : \vec{A} \to ok)] \vdash F : ok\end{array}}{K \vdash \texttt{void}\ p(\vec{A}\ \vec{x})\ S; V, F : ok}$$

[SIGNAL$_{\text{ENU-DEC}}$]
$$\frac{K[sig \mapsto Signal] \vdash V : ok \quad K \vdash \vec{k} : L\vec{I}T}{K \vdash \texttt{define Signal}\ sig\ : \vec{k}; V : ok}$$

[SIGNAL$_{\text{RANGE-DEC-INT}}$]
$$\frac{K[sig \mapsto Signal] \vdash V : ok \quad K \vdash k_1 : Int \quad K \vdash k_2 : Int}{K \vdash \texttt{define Signal}\ sig\ : k_1..k_2; V : ok}$$

[SIGNAL$_{\text{RANGE-DEC-FLOAT}}$]
$$\frac{K[sig \mapsto Signal] \vdash V : ok \quad K \vdash k_1 : Float \quad K \vdash k_2 : Float}{K \vdash \texttt{define Signal}\ sig\ : k_1..k_2; V : ok}$$

[DEVICE$_{\text{DEC-I}}$]
$$\frac{K[dev \mapsto Device] \vdash V : ok \quad K \vdash \vec{sig} : \vec{Signal}}{K \vdash \texttt{define Device}\ dev\ \texttt{input}\ : \vec{sig}\ ; V : ok}$$

[DEVICE$_{\text{DEC-O}}$]
$$\frac{K[dev \mapsto Device] \vdash V : ok \quad K \vdash \vec{sig} : \vec{Signal}}{K \vdash \texttt{define Device}\ dev\ \texttt{output}\ : \vec{sig}\ ; V : ok}$$

[DEVICE$_{\text{DEC-IO}}$]
$$\frac{\begin{array}{c}E[dev \mapsto Device] \vdash V : ok \\ K \vdash \vec{sig}_I : \vec{Signal} \quad K \vdash \vec{sig}_O : \vec{Signal}\end{array}}{K \vdash \texttt{define Device}\ dev\ \texttt{input}\ : \vec{sig}_I\ \texttt{output}\ : \vec{sig}_O\ ; V : ok}$$

[DEVICE$_{\text{DEC-OI}}$]
$$\frac{K[dev \mapsto Device] \vdash V : ok \quad K \vdash \vec{sig}_O : Signal \quad K \vdash \vec{sig}_I : Signal}{K \vdash \texttt{define Device } dev \texttt{ output } : \vec{sig}_O \texttt{ input } : \vec{sig}_I \; ; V : ok}$$

Table 5.32: Declaration type rules

In figure 5.33 the types rules for statements are described. For example the [ASSIGN] statement is only type correct, if the variable has the same type as the expression being assigned to the variable.

[DEVICE$_{\text{INIT}}$]
$$\frac{K[x \mapsto dev] \vdash V : ok \quad K \vdash dev : Device \quad K \vdash y : String}{K \vdash dev \; x = y \; ; V : ok}$$

[ASSIGN]
$$\frac{K \vdash x : A \quad K \vdash k : A}{K \vdash x{=}ex : ok}$$

[SKIP]
$$K \vdash \texttt{skip} : ok$$

[RETURN]
$$K \vdash \texttt{return} : ok$$

[PRINT]
$$\frac{K \vdash k_1 : A}{K \vdash \texttt{print } k_1 : ok}$$

[COMP]
$$\frac{K \vdash S_1 : ok \quad K \vdash S_2 : ok}{K \vdash S_1; S_2 : ok}$$

[IF]
$$\frac{K \vdash k_1 : Bool \quad K \vdash S_1 : ok}{K \vdash \texttt{if } (k_1) \; S_1 : ok}$$

[IF-ELSE]
$$\frac{K \vdash k_1 : Bool \quad K \vdash S_1 : ok \quad K \vdash S_2 : ok}{K \vdash \texttt{if } (k_1) \; S_1 \texttt{ else } S_2 : ok}$$

[WHILE]
$$\frac{K \vdash k_1 : Bool \quad K \vdash S_1 : ok}{K \vdash \texttt{while } (k_1) \; S_1 : ok}$$

[WAIT]
$$\frac{K \vdash k_1 : Bool}{K \vdash \texttt{wait } k_1 \; T : ok}$$

[FUNC-CALL]
$$\frac{K(p) = (\vec{x} : \vec{A_2}) \rightarrow A_1 \quad K \vdash \vec{e} : \vec{A_2}}{K \vdash p(\vec{k}) : A_1}$$

[FUNC-CALL$_{\text{VOID}}$]
$$\frac{K(p) = (\vec{x} : \vec{A}) \rightarrow Void \quad K \vdash \vec{k} : \vec{A}}{K \vdash p(\vec{k}) : ok}$$

[SET]
$$\frac{K \vdash d : Device \quad K \vdash s : Signal \quad K \vdash k_1 : A}{K \vdash \texttt{set}\ d\ s\ : k_1 : A}$$

[GET$_{\text{I}}$]
$$\frac{K \vdash d : Device \quad K \vdash s : Signal}{K \vdash \texttt{get input}\ d\ s : LIT}$$

[GET$_{\text{O}}$]
$$\frac{K \vdash d : Device \quad K \vdash s : Signal}{K \vdash \texttt{get output}\ d\ s : LIT}$$

[GET$_{\text{EMPTY}}$]
$$\frac{K \vdash d : Device \quad K \vdash s : Signal}{K \vdash \texttt{get}\ d\ s : LIT}$$

[BLOCK]
$$\frac{K \vdash V : ok \quad K_1 \vdash F : ok \quad K_2 \vdash S : ok}{K \vdash \{VFS\} : ok}$$

Where $K_1 = K(V, K)$
And $K_2 = K(F, K_1)$

Table 5.33: Statement type rules

In table 5.34 the type rules for the special keywords exceeds and deceeds are defined. Regardless if the exceeds is followed by an int or a float, it is evaluated to a boolean value.

[EXCEEDS$_{\text{INT}}$]
$$\frac{K \vdash k_1 : Int}{K \vdash \texttt{exceeds}\ k_1 : Bool}$$

$$[\text{DECEEDS}_{\text{INT}}] \quad \frac{K \vdash k_1 : Int}{K \vdash \texttt{deceeds}\ k_1 : Bool}$$

$$[\text{EXCEEDS}_{\text{FLOAT}}] \quad \frac{K \vdash k_1 : Float}{K \vdash \texttt{exceeds}\ k_1 : Bool}$$

$$[\text{DECEEDS}_{\text{FLOAT}}] \quad \frac{K \vdash k_1 : Float}{K \vdash \texttt{deceeds}\ k_1 : Bool}$$

Table 5.34: Exceeds and deceeds type rules

In figure 5.35 the types rules for the Pivot events are described. An example of this is the [$\text{WHEN}_{\text{SIGNAL}}$] is only in accordance to the type rules, if the device is of type device, the signal is of type signal and the enum is a literal value.

$$[\text{WHEN}_{\text{SIGNAL}}] \quad \frac{K \vdash dev : Device \quad K \vdash sig : Signal \quad K \vdash enum : LIT}{K \vdash \texttt{when}\ dev\ sig\ :\ enum : ok}$$

$$[\text{WHEN}_{\text{TIMEANDDATE}}] \quad \frac{K \vdash timeAndDate : D}{K \vdash \texttt{when}\ timeAndDate : ok}$$

$$[\text{WHEN-EXCEEDS}_{\text{INT}}] \quad \frac{K \vdash k_1 : Int \quad K \vdash S_1 : ok}{\texttt{when exceeds}\ k_1\ S_1 : ok}$$

$$[\text{WHEN-DECEEDS}_{\text{INT}}] \quad \frac{K \vdash k_1 : Int \quad K \vdash S_1 : ok}{\texttt{when deceeds}\ k_1\ S_1 : ok}$$

$$[\text{WHEN-EXCEEDS}_{\text{FLOAT}}] \quad \frac{K \vdash k_1 : Float \quad K \vdash S_1 : ok}{\texttt{when exceeds}\ k_1\ S_1 : ok}$$

$$[\text{WHEN-DECEEDS}_{\text{FLOAT}}] \quad \frac{K \vdash k_1 : Float \quad K \vdash S_1 : ok}{\texttt{when deceeds}\ k_1\ S_1 : ok}$$

[EVERY-START]
$$\frac{K \vdash k_1 : Int \quad K \vdash S_1 : ok \quad K \vdash timeAndDate : D}{K \vdash \texttt{every}\ k_1\ T\ \texttt{starting}\ timeAndDate\ S_1 : ok}$$

[EVERY-TIME]
$$\frac{K \vdash k_1 : Int \quad K \vdash S_1 : ok}{K \vdash \texttt{every}\ k_1\ T\ S_1 : ok}$$

Table 5.35: Event type rules

[INIT]
$$\frac{K \vdash S : ok}{K \vdash \texttt{init ()}\ S\ : ok}$$

Table 5.36: Init type rule

Chapter 6

Compiler and implementation

This chapter describes how the compiler for the Pivot language is implemented in accordance to the syntax design from chapter 4 and the semantics from chapter 5. The implementation of the compiler is described in detail and includes descriptions of abstract syntax tree generation and decoration as well as both java byte code generation and java code generation.

6.1 Compiler phases

The pivot compiler consists of several phases, which are illustrated in figure 6.2. First a parse tree is generated using the ANTLR tool. The ANTLR tool facilitates both the lexical analysis as well as the beginning of the syntax analysis. The parse tree generated by ANTLR, however, consists of many nodes, that are not needed for compilation. For example in the Pivot grammar, see section 4.3, when a variable of the type int is declared to be an int with the value 5, the value is actually nested within several nodes. The value 5 is inside the intVal node, inside the Atom node inside the expr node as seen in figure 6.1. All of these steps are not needed for the remaining phases of the compiler, so it was decided to create an abstract syntax tree with custom nodes, that contain only the necessary data. This abstract syntax tree then serves as a more efficient intermediate representation of the program. The abstract syntax tree is created using the visitor class generated by ANTLR, see more in section 6.3. After the AST is created, the syntax analysis concludes and sends the AST to the next step in the compiler, i.e. the Contextual analysis.

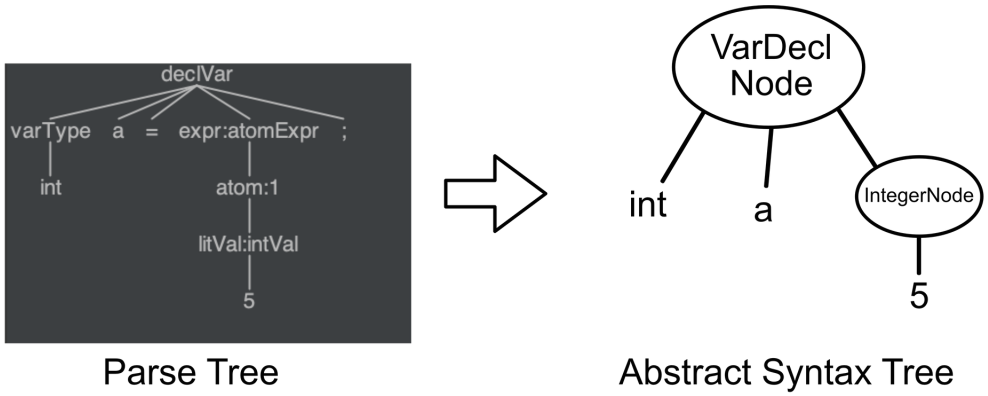

Figure 6.1: Variable declaration. Parse tree on the left and AST Node on the right

The contextual analysis is important in analysing the program in the context of the Pivot language. The syntax analysis for example cannot check if a variable is declared, since this is not a property of context free grammars, see page 129 in [25]. Since this is a property of the Pivot language, i needs to be checked at compile time.

The contextual phase consists of several visitors visiting the AST, see more about the visitor pattern 6.2. These visitors then decorate the AST with symbols and scopes in accordance with the semantics, see 5. One of these visitors is the very important TypeCheckerVisitor. This visitor checks at compile time, that all expressions are type correct and correspond to the typerules defined in 5.3.

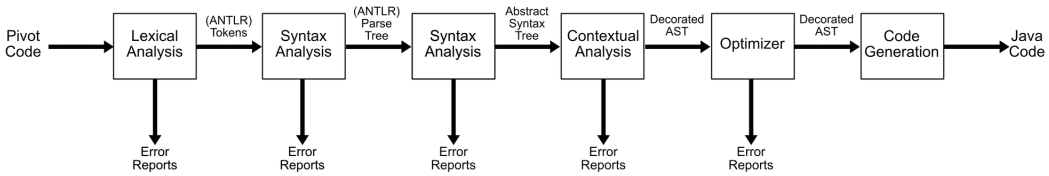

Figure 6.2: An illustration of the phases in the Pivot compiler

Finally the optimisation phase is reached. In this step the AST is optimised by evaluating expressions without variables. This should ensure faster runs times at the cost of slightly longer compile times.

The last phase is the code generation phase. In this phase a java program is generated from the decorated and optimised AST.

If any of these visitors at any point in time encounter an error, they will throw an exception, see section 6.1.1, with useful information for debugging. This information includes a descriptive error message as well as the line number in the Pivot

code, where the error occured.

6.1.1 Error handling

In order to enable easy debugging of Pivot programs going through the compiler several steps were undertaken. First, all nodes in the AST contain the method *getLineNumber()*. This method returns the line number where the node was declared in the Pivot program. This then becomes very useful in combination with the exceptions. All exceptions extend the custom CompileErrorException. This exception takes a message and the line number as arguments in the constructor. This means, that when the exception is thrown, the user can see both a custom message with an error message as well as the line in the Pivot code where the error was encountered.

6.2 Visitor pattern

Organising and managing the nodes in an abstract syntax tree for a programming language can be a difficult task. AST's for languages like Java contain around 50 node types and 200 phases[3], which needs to be handled in a systematic way. To craft the Pivot compiler, it was therefore decided to apply the modern software engineering principle called *Visitor pattern*[3], to manage the phase and node interactions in the compiler. The visitor pattern dictates, that the code for a phase should be written in a single class, and not distributed among various node types, and thereby create structure in the implementation.

An issue with most object-oriented programming languages, Java included, is that they use single dispatch to determine which visit method should be invoked. To handle this issue, the visitor pattern achieves a form of multiple dispatch for languages, that otherwise offer only single dispatch. It enables phase and node specific code to be invoked cleanly, while keeping all methods of a phase in a single class.

The basic principle of organisation for the visitor pattern in the different phases of the compiler, consists of the following:

- Every phase extends the base visitor class, so that it inherits a visit(AbstractNode n) method.
- Every node class has an accept function that accepts a visitor. This is to ensure, that the correct visitor is used. The accept function is invoked whenever it is wished to call the specific visit function found in the specific node class.

The visitor pattern is used in various different phases in the compiler. Every time an action or an analysis of the AST is needed, the visitor pattern is used to visit the different nodes in the tree and make changes.

It is also this functionality of keeping a phase inside a single class, that allows for more readable code, see for example code snippet 6.2.

The visitor pattern is firstly used to create the abstract syntax tree, then used to decorate the syntax tree, type-check the tree, optimise the tree and then used to generate code in java, which is the target language for the Pivot compiler.

6.3 Abstract Syntax Tree

As described in section 6.1 there is a lot of unnecessary data in the parse tree generated by the ANTLR parsing tool. For this reason it was decided, that an intermediate representation was needed. For this purpose an abstract syntax tree was designed. This was done by analysing all rules and tokens in the grammar. For example a variable declaration in the AST consists of a node containing a type, ID but then also an expression node. This contrasts the parse tree, where both the semi colon and equal sign is also present, but serve no purpose other than a syntactical one for the syntax analysis. The difference can also be seen in figure 6.1 for the variable declaration.

For creating the AST a visitor was created. This visitor extends the visitor provided by ANTLR called *PivotBaseVisitor<T>*. This is a generic visitor, that needs a type defined for the specific implementation. In the Pivot compiler an abstract class called Node was utilised, which the PivotBaseVisitor takes as type. The PivotBaseVisitor then contains methods for visiting all rules in the grammar and return an object of the type *Node*. All methods in the PivotBaseVisitor has a *context* argument, that contains the text and context of the grammar rule. These can be used for creating the nodes in the AST.

Extending the abstract class Node are several base nodes called *LeafNode*, *UnaryNode*, *BinaryNode*, *ListNode*, that implements some standard behaviour regarding printing and getting children of nodes with zero, one, two and *n* children. All of these nodes have the method *getChildren()*, which is needed for visiting the AST later on, but also for pretty-printing the AST, which is important in the debugging phase of creating the ASTBuilderVisitor. The very first node is of the class *ProgramNode* which extends *UnaryNode*.

The ProgramNode only contains references to its child node and the Symbol Table. The Symbol Table is populated in the contextual analysis phase of the compiler, also know as the decoration of the AST.

6.3.1 Symbol Table

When decorating the AST, every new symbol that is visited in the syntax tree, needs to be added to the symbol table, see more in section 6.4. The symbol table is also responsible for tracking which declaration is in effect when a reference to the

symbol is encountered. To implement a symbol table, we need to consider which type of scoping the Pivot language uses, how to store the symbols in the symbol table and also decide whether to have one big symbol table or a symbol table with nested symbol tables for each scope.

When implementing the symbol table, the simplest method was to create a data structure of the Java type *arraylist* containing all the declared symbols within the current scope, and then whenever a reference to the symbol is met, the symbol list would then be searched through and checked if the symbol is already declared. It was decided to use this basic implementation, although a few changes had to be made in order for the symbol table to be working fully. Since it should be possible for functions and variables to have the same name id, the symbol list implementation had to be modified. Instead of having one list containing all id's in the scope, the function id's were put in a separate list, and since functions can only be declared in the global scope, it makes sense to split the list of symbols in two.

Firstly, a new symbol table is made for the global scope in the program. Here, all global variables, devices and signals will be declared, the *init*() function and all the events in the pivot code are also contained within the global scope. For every block(scope) e.g event, function, iteration or selection structures, a new nested symbol table is made to contain the local variables and declarations within the current scope.

To handle the main interactions within the symbol table, the following functions were made for the visitors:

- *openScope*() For every new block, openScope() is called.
- *closeScope*() Whenever a block ends, closeScope() is called.
- *addSymbol*(*symbol*) When a new symbol, that isnt contained within the current symbol table is declared, addSymbol(symbol) is called.
- *retrieveSymbol*(*symbol*) To get a specific symbol from the symbol table, retrieveSymbol(symbol) is called, which returns the symbol that is put as the function parameter.
- *idExistsInScope*(*id*) Checks if an id for a symbol already exists in the current scope. If it is, idExistsInScope(id) returns true, and else false.

```
1  private Optional<Symbol> getSymbol(String id){
2      // Check every symbol within this scope
3      for(Symbol s : localSymbols) {
4
5      // Check the literals defined in signal symbols
6      if (s instanceof SignalTypeSymbol) {
7          Optional<FieldSymbol> matchingSignalLiteral =
8          ((SignalTypeSymbol) s).getSignalLiteral(id);
9          //(at return) The extra Optional.of converts
           Optional<FieldSymbol> to Optional<Symbol>
10         if(matchingSignalLiteral.isPresent()){
11             return Optional.of(matchingSignalLiteral.get());
12         }
13     }
14
15     // Check if the symbol id matches the given id
16     if (s.id.equals(id))
17         return Optional.of(s);
18     }
19
20     // Check the parent block if no local symbols match the id
21     if(hasParent())
22     return parentBlock.getSymbol(id);
23
24     return Optional.empty();
25 }
```

Code snippet 6.1: Implementation of get symbol in the symbol table

To look closer at the implementation of the getSymbol() function, the function return type is Optional<Symbol>. This is because there is a possibility that the id we are searching for in the current scope, is not contained within the symbol table. For each symbol in the local scope, it is initially checked if the symbol is an instance of SignalTypeSymbol. If true, the id for the signal is found and put into the Optional<FieldSymbol> matchingSignalLiteral variable. It is then checked if the matchingSignalLiteral is present in the symbol table. If there is a value present the id is returned.

If the symbol is not of type SignalTypeSymbol, the symbol id is checked regularly with the use of the .equals() function.

Lastly, if there is no local symbol that matches the id, the function is called recursively on the parent scope. If no matching symbol is found in all parent scopes, an empty optional is returned.

6.4 Decorated abstract syntax tree

This section discusses the contextual analysis phase of the compiler. In this phase the abstract syntax tree is decorated using the symbol table, that contains all the

scopes and variables. This phase is executed by using several visitors extending the *ASTBaseVisitor* class, so they have a visit method for each node class.

The abstract syntax tree is visited by using the code in code snippet 6.2. By using the accept method, it is made sure, that only visitors of the type *ASTBaseVisitor* can actually visit the method. All four visitors in code snippet 6.2 extend the *ASTBaseVisitor*.

```
1  private static void decorateAST(Node ast){
2      // Add functions to symbol table
3      ast.accept(new FunctionVisitor());
4      // Add all variables to symbol table
5      ast.accept(new DeclarationVisitor());
6      // Assign types to all expressions
7      ast.accept(new TypeAssignmentVisitor());
8      // Type check all statements
9      ast.accept(new TypeCheckerVisitor());
10 }
```

Code snippet 6.2: The method in the compiler to decorate the AST

6.4.1 Declaration visitors

The first visitors dispatched to the AST are the *FunctionVisitor*, *DeclarationVisitor* and *TypeAssignmentVisitor*. The FunctionVisitor inserts all functions in the SymbolTable inside the FunctionSymbol. The FunctionSymbol includes some standard behaviour for all functions, including *GetParameters()* and *getReturnType()*, see section 6.3.1 for more details about the symbol table. The *TypeAssignmentVisitor* assigns types to all expressions.

6.4.2 Type Checking

Since the Pivot language is statically typed, it is possible to typecheck all expressions at compile time, since no expression or variable can change type during run time, see section 4.1.2 for the design and section 5.3 for type system. This gives Pivot an important property, namely that if all expressions, variable declarations and assignments are correctly typed and this is checked at compile time, then these statements cannot introduce a run time error due to typing[10]. This can greatly increase the reliability of a Pivot program, which was also an important goal found in the language qualities analysis, see section 3.3.

In order to achieve this, two visitors were introduced in the compiler. The first visitor is called *TypeAssigmentVisitor*, see section 6.4.2, and the second the *TypeCheckerVisitor*. Both of these visitors visit AST in the contextual analysis part of the compiler. This is possible, since both visitors extend the *ASTBaseVisitor* class.

The *TypeCheckerVisitor* was implemented after creating the type rules system of pivot, see section 5.3. One example of this approach is described in the following

example.

$$[\text{VAR}_{\text{DEC}}] \qquad \frac{K[x \to A] \vdash V : ok \qquad K \vdash ex : A}{K \vdash A\ x = ex; V : ok}$$

This type rule describes, that a variable x is declared with the type A. The variable is then at the same time initialised to an expression. The preconditions then says, that this is only *ok*, if the variable x is bound to the type A in the type environment, and the expression is also of the type A.

The first precondition is ensured by the symbol table. The *DeclarationVisitor* inserts the variable with a type into the symbol table. The expression gets its type assigned to it by the *TypeAssigmentVisitor*. Lastly TypeCheckerVisitor runs the following code seen in code snippet 6.3.

```
1 @Override
2 public Void visit(VarDeclNode node) {
3     // If the expression does not have the same type as the variable being
       declared, throw exception.
4     if(!node.getExpr().getType().equals(node.getVarType())){
5         throw new ExpressionWrongTypeException("Expression has more types
      in it or doesn't match variable type.", node.getLineNumber());
6     }
7
8     return super.visit(node);
9 }
```

Code snippet 6.3: TypeCheckerVisitor visits a variable declaration

The *TypeCheckerVisitor* compares the type of the expression to the type of the variable being declared. If they are not equal, an exception is thrown with an error message to the programmer.

Another example of the *TypeCheckerVisitor* that is highlighted here, is the function call. This statement is a bit more difficult to check the type of, since it does not only need to check the type of one expression, but of for an arbitrary amount of arguments given to the function.

$$[\text{FUNCTION-CALL}] \qquad \frac{K(p) = (\vec{x} : \vec{A_2}) \to A_1 \quad K \vdash \vec{k} : \vec{A_2}}{K \vdash p(\vec{k}) : A_1}$$

This type rule says, that the function only evaluates to type A1, if the function in our type environment is declared to return A1. Also the function call is only valid, if for all formal parameters x that have type A2, then each actual parameter, or argument, must also be of type A2. This was implemented in the *TypeCheckerVisitor*

and can be seen in code snippet 6.4.

```
@Override
public Void visit(FuncCallNode node) {
    // Get the function
    Optional<FunctionSymbol> sym = st.getFunctionSymbol(node.getID());
    if(sym.isPresent()) {
        FunctionSymbol funcSym = sym.get();
        // Check that the correct number of variables are parsed
        if (funcSym.getParameters().size() == node.getArguments().size())
    {
            // Check that all variables have the same type
            for (int i = 0; i < node.getArguments().size(); ++i) {
                // Every argument (an expr) should have the same type as
                // the formal parameters gotten from the funcSym.
                // The order of the parameters MATTERS.
                if(!((ExpressionNode)node.getArguments().get(i)).getType()
                      .equals(funcSym.getParameters().get(i).getTypeID())){
                    throw new ArgumentWrongTypeException(...);
                }
            }
        } else{
            throw new IncorrectArgumentAmountException(...);
        }
    } else {
        // if the function is not present in the symbol table, throw
    exception.
        throw new FunctionNotDeclaredException(...);
    }

    return super.visit(node);
}
```

Code snippet 6.4: TypeCheckerVisitor visits a function call

First the function symbol is pulled from the symbol table. If the function is not present, an exception is thrown. Next a check is made, if the function call has the same amount of variables as the function declaration. This check was introduced, since there is no reason to check the types of the function call, if an argument is missing. This therefor serves as a kind of short circuiting in the compiler to improve performance. This way a better error can also be reported. In this case an *IncorrectArgumentAmountException* is thrown. If the function is present in the symbol table and the amount of arguments parsed by the function call the type checking of the arguments can commence. A for loop is used to go through each argument to check the type. Since the order of the arguments matters in the Pivot programming language, they are checked from 1 to n. If any one of the arguments has the wrong type, an exception is thrown with an error message for the programmer. The programmer is also told which exact argument has the wrong type. This was done to make debugging easier. The last line visits the children of

the function call, which includes the arguments, which are then also type checked. The arguments consist of expressions, which can also include function calls.

TypeAssigmentVisitor

This visitor has the purpose of assigning types to all expressions. These expressions include setters, getters, function calls, variables and lastly the three primitive types, string, float and int. In the AST, all of these statements are represented by a node, that extends the *ExpressionsNode* abstract class. This class has one field with a string called type and some getters and setters for this type. This means, that all of the above mentioned expression can be assigned a type in the form of a string.

The *TypeAssigmentVisitor* then uses the functionality of the abstract class *ExpressionsNode* to assign types to all expressions. An example of how an expression can be recursively visited to make sure, that it is type correct is shown in the code snippet 6.5.

```
1  @Override
2  public String visit(AddExprNode node) {
3      String leftNodeType = visit(node.getLeftChild());
4      String rightNodeType = visit(node.getRightChild());
5
6      if(!leftNodeType.equals(rightNodeType)){
7          throw new ExpressionWrongTypeException("Type mismatch expected: '"
       +
8                  leftNodeType +
9                  "' got: '" +
10                 rightNodeType +
11                 "'"
12                 , node.getLineNumber());
13     }
14
15     node.setType(leftNodeType);
16     return node.getType();
17 }
```

Code snippet 6.5: TypeAssignmentVisitor visits and addition expression

This method first visits both sides of an add expression and finds the type. An example of this could be expression **2 + 3 - 3**. Since all three are individually represented by an IntegerNode in the AST, the visit method for each would return the string value *int*. The **3 - 3** part of the expression would be visited recursively inside the *visit(node.getRightChild())*, which then visits and AddExprNode again.

Following this, an if statement checks if the left node and right node have the same type. If this is not the case an exception is thrown. If, however, this is the case, the entire expression will have its type set to be the same as the two children. Next the type is returned.

Another almost identical visitor method is made for the multiplication nodes in the AST.

Since function calls can also return values and be part of an expression, they must be type checked as well. This means, that they must also have a type assigned to them, see code snippet 6.6.

```
@Override
public String visit(FuncCallNode node) {
    Optional<FunctionSymbol> funcSymbol = st.getFunctionSymbol(node.getID
    ());
    if(funcSymbol.isPresent()){
        node.setType(funcSymbol.get().getReturnType());
    } else {
        throw new FunctionNotDeclaredException("Function '" +
                node.getID() +
                "' not declared.",
                node.getLineNumber()
                );
    }

    // Make sure to visit all arguments to assign types to them as well.
    super.visit(node);

    return node.getType();
}
```

Code snippet 6.6: TypeAssignmentVisitor visits a function call

This visit method first fetches the function information from the symbol table. If the function is not declared, an exception is thrown. If the function is declared, the function call node gets its type assigned to the type declared inside the function in the function table. After this process *super.visit(node)* is called. This makes sure, that all the arguments of the function call are also visited and assigned types. Lastly, the function calls type is returned as a string.

6.4.3 Optimisation

In order to achieve faster execution, some optimisation can be useful. In the Pivot compiler, there is only one type of optimisation, it is done with the OptimiseExprVisitor, which visits the AST. The OptimiseExprVisitor has functionality to shorten expressions by evaluating some expressions already at compile time. This is done for expressions consisting of only literal string, float or int values. Since these do not contain any variables, the value of the entire expression can be known at compile time. In the code snippet 6.7 the code for the optimiser visitor function inside the OptimiseExprVisitor. This visit method evaluates the value of an expression if the expression has no variables, and all parts of the expression are of the same type, i.e string, float or int.

```java
@Override
public Object visit(MultiExprNode node) {
    ExpressionNode newNode;

    var left = visit(node.getLeftChild());
    var right = visit(node.getRightChild());

    // Check if both are integer
    if(left instanceof IntegerNode && right instanceof IntegerNode){
        // Call recursively until a new IntegerNode is returned.
        if(node.getOperator() == Operator.MULTPLY){
            Integer newValue = (((IntegerNode) left).getVal() *
                                ((IntegerNode) right).getVal());
            newNode = new IntegerNode(node.getContext(), newValue.toString());
            return newNode;
        } else if (node.getOperator() == Operator.DIVIDE){
            Integer newValue = (((IntegerNode) left).getVal() /
                                ((IntegerNode) right).getVal());
            newNode = new IntegerNode(node.getContext(), newValue.toString());
            return newNode;
        }
    }
    // Check if both are float
    if(left instanceof FloatNode && right instanceof FloatNode){
        // Call recursively until a new FloatNode is returned.
        if(node.getOperator() == Operator.MULTPLY){
            Float newValue = (((FloatNode) left).getVal() *
                              ((FloatNode) right).getVal());
            newNode = new FloatNode(node.getContext(), newValue.toString());
            return newNode;
        } else if(node.getOperator() == Operator.DIVIDE){
            Float newValue = (((FloatNode) left).getVal() /
                              ((FloatNode) right).getVal());
            newNode = new FloatNode(node.getContext(), newValue.toString());
            return newNode;
        }
    }

    return super.visit(node);
}
```

Code snippet 6.7: Multiply or divide expression optimiser

First the function checks if both sides evaluate to an IntegerNode, meaning to an integer value. If this is the case it checks if the operator is either multiply or divide. The value is then returned as a new IntegerNode with the correct value inside of it. The entire old expression is then replaced with this new Node in the AST. See the code for the replacement in code snippit 6.8. When the expr is visited, it either returns the new optimised expr or null.

This optimisation is for example used in the variable declarations. If the optimiser finds a node of the type ExpressionNode, hence not null, this new expression is set in place of the old one.

```
1   public Object visit(VarDeclNode node) {
2        var newExpr = visit(node.getExpr());
3        if(newExpr instanceof ExpressionNode){
4            node.setExpr((ExpressionNode)newExpr);
5        }
6
7        return super.visit(node);
8    }
```

Code snippet 6.8: Optimise variable declaration

An illustration of this optimisation inside a variable declaration can be seen in figure 6.3.

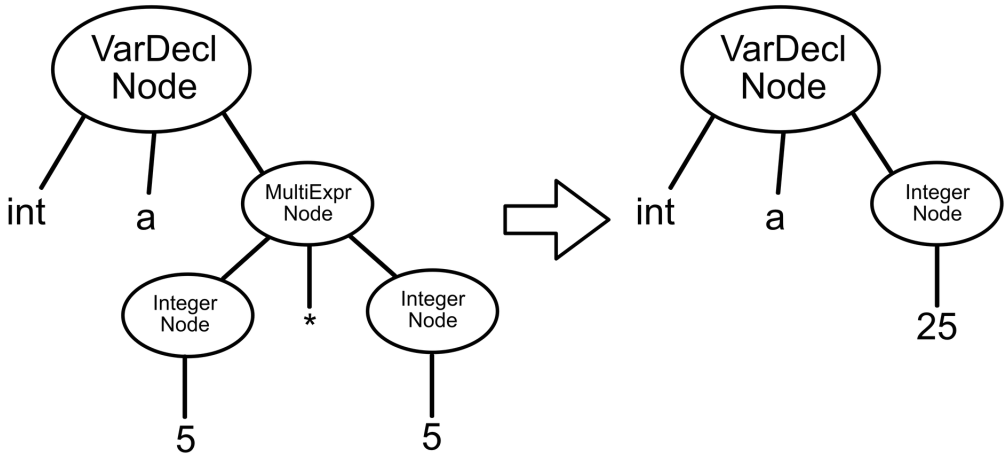

Figure 6.3: An illustration of the Pivot compiler optimisation

A similar function was created for the plus and minus expressions. In the plus and minus expressions, however, strings are also present. In the pivot language it is possible to concatenate strings in the same way, as in the Java language, by using the plus operator. For example:

"string1" + "string2" = "string1string2"

It is, however, not possible to subtract to strings. The compiler will show an error message if the user tries to divide, multiply or subtract from or to strings.

This expression optimisation should result in faster execution times on a machine running the program, since some machine instructions for the Java just-in-time compiler compiling the expression, as well as the evaluation the value of the expression, can be left out. This will, however, increase the compile time slightly,

because of the extra visitor.

6.5 Java Code generation

The code generation phase begins when the abstract syntax tree has been created, decorated, and possibly optimised, and all compile time error checks have been made. This phase consists of outputting code in the target language, in this case java code, based on the decorated abstract syntax tree and symbol table.

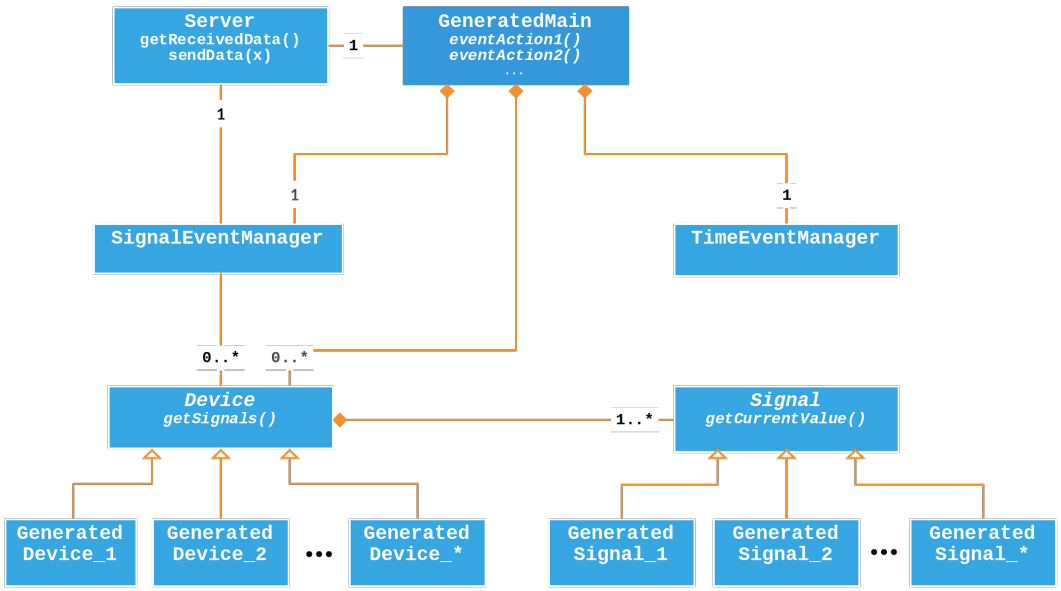

Figure 6.4: Class diagram describing the classes generated by the compiler

The output code consists of a combination of predefined classes and generated code. Figure 6.4 shows the class diagram for the output code generated by the pivot compiler. In this diagram, classes with the 'Generated' prefix are classes which are generated from scratch, and their contents depend entirely on the pivot source code.

When handling events, the code differentiates between signal-based events and time-based events. The *TimeEventManager* and *SignalEventManager* classes are responsible for correctly triggering events by calling the *eventAction()* methods in main. Event management is discussed further in section 6.5.4.

6.5.1 Class builder class

When building the generated classes, the *ClassBuilder* class is utilised. This class also helps make the code in the code generating visitors more readable, since they

can simply invoke methods from this class to generate the code. The *ClassBuilder* class uses the builder pattern. This was used, since a constructor for a new class would otherwise take too many arguments, which would become difficult to read. Also some arguments would not be relevant for all classes to be generated. This would mean, that the constructor of the *ClassBuilder* would have to be overloaded many times, further decreasing readability. For this reason, the builder pattern is preferred. The visitor method invoking the *ClassBuilder* can be seen in code snippet 6.9.

```
@Override
public ClassBuilder visit(DeclsNode node) {
    for (Node n : node.getChildren()) {
        ClassBuilder classBuilder = visit(n);
        if (classBuilder != null)
            JavaFileWriter.writeClass(classBuilder);
    }

    return null;
}
```

Code snippet 6.9: Invokation of the ClassBuilder

If the an instance of the *ClassBuilder* is returned from the visit method, it is then written to the generated code module with the *JavaFileWriter* class.

An example of the use of the *ClassBuilder* to generate something inside the class is seen in the code snippet 6.10, where a get method is prepared for generation.

```
public ClassBuilder appendGetMethod(String type, String varName){
    codeBuilder
    .append("public ")
    .append(type)
    .append(" ")
    .append("get")
    .append(varName);

    codeBuilder
    .append(START_PARAN)
    .append(END_PARAN);

    openBlock(BlockType.METHOD);
    appendReturnStatement(varName);
    closeBlock(BlockType.METHOD);

    return this;
}
```

Code snippet 6.10: Using the ClassBuilder to build the a get method for a class

First the Java keyword public is added, so that the generated getter method can be accessed. Next the type is appended as well as the word get and the name of

the variable name, that the getter should get. Next a block is opened. This simply means, that the curly brackets are added. Next the return statement is added. Finally, the block is closed and the *ClassBuilder* returned. The code in code snippet 6.10 is also a good example of how the builder pattern can increase readability, and therefor maintainability, of the Pivot compiler.

The *GeneratedMain* class contains the methods that should be executed when each event is triggered. Furthermore, it contains the *init* method and functions defined in the pivot source code. Signal and device definitions are translated into classes in the generated java code. These classes inherit from the predefined abstract classes *Device* and *Signal*. A signal class has a type, such as *Integer* or *String*, and variables of that type corresponding to the values in the signal definition. A device contains one or more signal objects corresponding to the valid input and output signals for the device.

6.5.2 Generating Signal and Device classes

As mentioned in earlier, the custom generated signal and device classes all inherit from the abstract Signal and Device class respectively. The *Device* and *Signal* classes are generated by visiting the declaration nodes in the AST. As an example, by visiting a signal declaration node for a 'toggle' signal definition, a the following signal class declaration is emitted:

```
1 public class Toggle extends Signal <Integer> {
2     public Toggle() {
3         // Set the default value to 0
4         super.setCurrentValue(0);
5     }
6 }
```

Code snippet 6.11: Signal class generated from a signal declaration node

As previously mentioned this class extends the predefined *signal* class which is the super class of all generated signal types. Then, by visiting the children of the signal declaration node, the contents of the class are emitted. The signal node might contain signal enum nodes (representing a signal value such as off=1), which when visited emits a variable declaration corresponding to the name and value of the signal enum.

```java
public class Toggle extends Signal <Integer> {
    public final int Off = 0;
    public final int On = 1;
    public Toggle() {
        // Set the default value
        super.setCurrentValue(0);
    }
}
```

Code snippet 6.12: Signal class definition with variable declarations corresponding to the signal's values

Device definitions are generated very similarly; a class definition is emitted when visting a device definition node, and the contents of the class are emitted when visiting its child nodes. Device classes contain signal variables for each type of signal they can receive and send, as well as a getter method for each of these signals.

Getter Method

When the *getCurrentValue()* is called on a signal in a device, it will return the last signal sent/received depending on whether the signal is an output or input signal. If no prior signal has been sent it returns a value based on the first defined signal type for that device. For example if a signal has its values defined in pivot code as toggle Off = 0 and On = 1, then the default value is the value of *off* which is zero.

6.5.3 Generating the main class

The bulk of the generated code resides in the Main class. This class contains all variable declarations, regular functions and event functions translated from the Pivot source code. Furthermore, it contains some standard functions for initialising the server and event managers.

6.5.4 Event management

Events are scheduled and executed automatically by event managers which are included when compiling pivot code. These managers are responsible for starting and stopping event execution threads based on the conditions specified in the Pivot event definitions.

As mentioned and motivated in section 4.1.6, only one thread can run per event definition. This could pose a problem since an event might still be running when it is activated again. Simply waiting for the old thread to finish running is not an option, because there is no guarantee that the event will ever terminate due to the inclusion of potentially infinite while loops. Furthermore, forcefully killing a thread is dangerous in Java, because it can cause so-called damaged objects and thereby unintended behaviour[20].

Instead the threads' interrupt flag is set to true and the generated code checks this flag at the start of every loop iteration and function call. A thread will then peacefully stop itself upon discovering that its interrupt flag has been set to true.

In conclusion, if an event is restarted before the current event thread has stopped executing, the manager will set the interrupt flag and wait for the thread to terminate before starting the new event thread.

Event concurrency

As previously mentioned, Pivot allows execution of events in parallel. These events may modify globally scoped variables, potentially leading to a race condition where multiple events are accessing the same variables at the same time, which can cause unintended behaviour. Recall that the semantics for locking shown in table 6.3 describe a solution where events use 'synchronised blocks'. Inside these blocks statements must obtain locks on all the variables they use, before they can be executed. A more detailed description of these semantics can be found in section 5.2.14.

[SYNC-BLOCK-1]
$$env_v, env_f \vdash < S_1; SB,\ sto,\ lock > \Rightarrow \\ < S_1; SB,\ sto,\ lock[v \mapsto (S_1)] >$$

$$\textbf{if } nonlocked \neq \varnothing \ \ and\ lock\ v = \mathit{ff} \\ \textbf{where } v = min\ (nonlocked) \\ nonlocked = \{x\ \in\ glo[\![S_1]\!] \mid lock\ x \neq (S_1)\}$$

Table 6.3: Lock semantics

In the implementation this is accomplished by first finding all the global variables used in a statement. A class called *GlobalVarVisitor* is responsible for finding and returning all global variables used in a given node. It does so by recursively visiting child nodes and finding all variable nodes, then looking them up in the symbol table to determine if they are defined in the global scope.

The semantics also state that locks are always acquired in the same order to avoid deadlocks. In the implementation, the *GlobalVarVisitor* accomplishes this by sorting the set of variables according to the linenumber they were defined at.

```
Comparator<FieldSymbol> lineComparator
            = Comparator.comparingInt
            (fieldSymbol ⟶  fieldSymbol.getLineNumber());

    public SortedSet<FieldSymbol> visit(StmntNode stmnt){
        TreeSet<FieldSymbol> globalSymbols
                = new TreeSet<>(symbolComparator);

        // Find all global symbols in child nodes
        globalSymbols.addAll(visit(node.getChildren()))
    }
```

Code snippet 6.13: Find all global variables in a statement

Using this sorted set of variables, locks are generated using java's *synchronized* keyword. Code snippet 6.14 shows some generated java code where locks are attained on the global variables used in the statement $c = b + a$

```
    Integer c;
    Integer a;

    public void sampleEvent () {
        Integer b;

        // Lock c first because it was defined first
        synchronized (c){
            synchronized (a){
                c = b + a;
            }
        }
    }
```

Code snippet 6.14: Generated java code with locks for global variables

Signal event manager

Signal events are triggered by incoming signals, sent by devices on the network. The server handles incoming signals by inserting the received data into a linked blocking queue which is accessible by the signal event manager. The signal event manager continuously polls the signals from this queue and parses them to determine if any events should be executed. If the event manager polls the queue when it's empty the event queue will be blocked until a new signal is added, which is a feature of the Java class LinkedBlockingQueue.

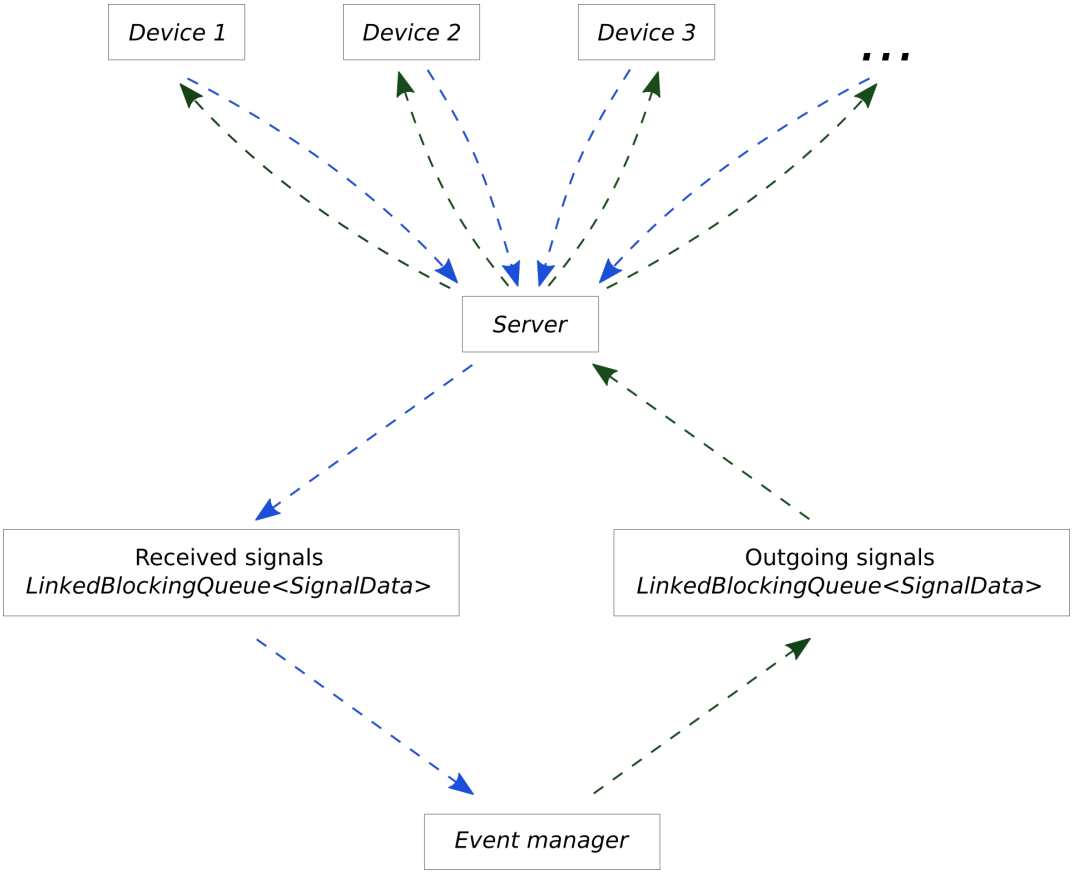

Figure 6.5: Signal data flow between event managers and devices. Device output signals are shown as blue arrows and device input signals as dark green arrows.

Code snippet 6.15 shows how the signal event manager handles signals by triggering the appropriate events and then waiting for new signals.

```
public void run(){
    while(running){
        // Retrieve a signal from the signal queue
        // take() automatically sleeps until a signal is availble
        SignalData signalData = signalQueue.take();

        triggerEvents(signalData);

        // Update the current value variable
        // to correspond to the received signal
        updateCurrentValue(signalData);
    }
}


private void triggerEvents(SignalData signalData){
    for(SignalEvent event : signalEvents)
        if(event.satisfiesCondition(signalData))
            event.executeEvent();
}
```

Code snippet 6.15: The loop handling incoming signals by executing the appropriate events and updating the current value

When the signal event manager takes a signal from the queue, it first checks if a signal triggers any events by comparing their event conditions to the received signal. A separate event thread is launched for each triggered event. This corresponds to the semantics shown in table 6.4, where incoming signals are compared to each event condition, and triggered events are launched in parallel. Next the current signal value of the appropriate device is updated according received signal data. This also corresponds to the semantics, where the variable environment is updated to reflect the received signal.

$$\text{[TRIGGER-1]} \quad env_e,\ env_v \vdash < trigger(E_1,\ E_2 ... E_n) > \Rightarrow < \{|trigger(E_2 ... E_n),\ E_1|\} > \quad \textbf{if } env_v,\ sto \vdash eb \Rightarrow \text{tt} \quad \textbf{where } env_e\ E_1 = (SB_1,\ eb_1,\ env'_v,\ env'_f)$$

Table 6.4: Event Trigger

Time-based event manager

Events can be defined to be repeated with a specified time interval. When the next execution time is scheduled from the moment the event starts executing, and thus not from when the event has finished executing. Defining the time in which the event should be rescheduled can be done using the **every** keyword, followed by a

time interval.

```
@Override // The main loop responsible for executing and rescheduling
    events when needed
public void run(){
    while (true){
        if(eventExecutions.isEmpty()) return; // Return if there's no more
     events left to be executed

        TimeEvent nextEvent = eventExecutions.first();

        // Sleep until the next event is ready to be executed
        waitUntil(nextEvent.getNextExecutionTime());

        // Execute the event action and remove it from the queue
        nextEvent.executeEventThread();
        eventExecutions.remove(nextEvent);

        // Reschedule event and add it back in if it's repeatable
        if(nextEvent.isRepeated()){
            nextEvent.rescheduleEvent();
            eventExecutions.add(nextEvent);
        }
    }
}
```

Code snippet 6.16: The loop handling the time-based events.

The code in code snippet 6.16 is the run method inside the *TimeEventManager* class. This thread should never terminate if time based events are defined, therefor it contains a *while(true)* loop. The TimeEventManager is responsible for triggering all events, that are based on time. First the thread checks if the set of eventExecutions is empty. If the evenExecutions set is empty, this means, that no time based events have been defined in the Pivot program. Then the method simply returns, which terminates the thread. If, however, some time based events are defined, the first event is pulled from the set of events. In order to save CPU cycles in the system running, the waitUntil method was introduced. This method simply calls sleep on the thread until the next time based event should be executed. After this the next event is executed and removed from the set of events. If the event is a repeated event, i.e. using the *every* keyword in the Pivot code, the event will be rescheduled and added back to the set of events. This ensures that all events are triggered correctly, but at the same time avoids wasting CPU cycles by sleeping the thread until the next event execution time. This may enable less powerful devices to act as server in a program written in Pivot.

6.6 Java byte code generation

Since the target language of the Pivot compiler is Java, and Java is compiled to Java Byte Code before execution, this begs the question of whether or not, Pivot should be compiled directly to Java Byte Code. This would still allow for it to function in the same domain as described in section 3.2.1. In order to get the Pivot code compiled to Java Byte Code a popular tool called Jasmin can be used.

Jasmin is an assembler for the Java Virtual Machine. It takes assembly like instructions and converts them to into binary Java class files[15]. This means that in order to convert the Pivot code into Java Byte Code, it must first be converted into Jasmin code. This was done using the Visitor Pattern. This was done by creating a new visitor for the abstract syntax tree, where each visit method generates code in accordance to the Pivot code. The current and final implementation of the Jasmin compiler for the Pivot language supports variable declaration and assignment, addition and subtraction and if/else statements. The Pivot program that was compiled to Jasmin code is very simple:

```
1  init(){
2      int a = 10;
3      if(1 == 2){
4          print a;
5      }else{
6          print 2;
7      }
8      int c = a+10;
9      if(c != a){
10         print c;
11     }else{
12         print a;
13     }
14 }
```

Code snippet 6.17: Pivot example program for byte code generation

The Jasmin compiler also supports the other boolean operators. The code that the JasminVisitor emits, can be found in the appendix. In order to assign variables, the *ldc* instruction, which pushes a constant on to the stack, is used. Then if a number should be stored as a local variable, the *istore* instruction is used. The *istore* instruction only works for integers, but currently, the JasminVisitor only supports integers. This assignment happens on line 2 in code snippet 6.17 and the translation can be seen below:

```
1 //Push 10 onto the stack
2 ldc 10
3 //Store the topmost element of the stack at location "0"
4 istore 0
```

Code snippet 6.18: Storing a local variable in Jasmin

If the purpose was instead to add two numbers together and then store them in a new local variable, like at line 8 in code snippet 6.17, the code would instead be as in code snippet 6.19

```
1 //Load the integer stored at location 0
2 iload 0
3 //Push 10 onto the stack
4 ldc 10
5 //pop the two topmost elements of the stack, add them together and push
     result to the top of the stack
6 iadd
7 //Store the topmost element of the stack at location "1"
8 istore 1
```

Code snippet 6.19: Addition in Jasmin

When creating an if/else statement, the approach is similar. For instance, the if/else statement at lines 3-7 are translated to the following:

```
1 //Push 1 and 2 to the stack before the comparison
2 ldc 1
3 ldc 2
4 //Then compare them using ifeq L1. The defined label is where the
     execution should jump to if the condition is true.
5 ifeq L1
6 //If the condition is not true, we jump to the else block
7 goto L0
8 //Label L1: ie the true block
9 L1:
10 //code
11 //when execution of this block is finished, we jump to the "end" label L2
12 //Label L0: ie the false block
13 L0:
14 //code
15 //after executution we dont have to jump, since we're already at the end
     label
16 L2:
17 //code
```

Code snippet 6.20: If/else statements in Jasmin

The labels and the locations of the variables is all handled in the JasminVisitor class. Whenever a variable is declared, the variable name is stored in a Map. The key is the name and the value is the size of the map at the time of declaration. This means, that the variable is then assigned to the correct location in storage.

Translating an entire Pivot program to Jasmin would require lots of effort and would not necessarily provide any significant benefits. Jasmin provides the ability to optimise a lot, since it is very low level, but the ease of use is not very good. This combined with the fact that the compiler that ships with the JDK, more precisely the JavaC and JIT compiler, make a lot of optimisations [11], makes it harder to argue for compiling directly to Java Byte Code. For this reason it was decided to translate to Java and let the JavaC compiler compile to Java Byte Code.

Chapter 7

Testing

In this chapter it will be discussed how the Pivot language and compiler are tested. This includes both unit tests as well as system tests. These tests were made to reduce the risk of unintended behaviour in the compiler and the generated code.

7.1 Unit tests

Unit tests are used to test, that individual components of the Pivot compiler are running as intended. This is done by using the JUnit library to assert the expected results.

7.1.1 Time Comparison

Since one of the major features of the Pivot language is time comparison, which means if an event is set to trigger at a specific time, Pivot will compare the current time to the time triggering the event. For this, some tests needed to be conducted to ensure correct behaviour.

The time comparisons in the code generated by the Pivot compiler is done with *isComparisonTrue()*, which returns a boolean, inside the *Utils* class. The time comparison was then tested by using this method and asserting the correct outcome. This was, however, complicated since the outcome of this method depends on the time, when it is run. For this reason, it had to be taken into account in the tests. An example of this can be seen in code snippet 7.1.

```
@Test
void isComparisonTrueTest04(){
    int curResult = LocalTime.now().compareTo(LocalTime.of(11,10));

    // Generated from 'now < 11:10'
    boolean bool = Utils.isComparisonTrue(LocalDateTime.now(),
    ComparisonOperator.SMALLERTHAN, LocalTime.of(11, 10), false);

    // If time of running before 11:00
    if(curResult < 0) assertTrue(bool);
    // If time of running after 11:00
    else if (curResult > 0) assertFalse(bool);
    // If time of running is 11:00. Should still be false.
    else assertFalse(bool);
}
```

Code snippet 7.1: Test for time comparison

First the current result depending on the time running the tests is run. In the example in code snippet 7.1 the comparison tests, if the time **now** is smaller than 11:10, meaning that it is earlier in the day, than 11:00. Next the boolean value from the tested method is found. Then depending on the current time, the result is asserted to be either true or false.

7.1.2 Compiler

Several unit tests were conducted for the Pivot compiler. The approach when developing these, was to initially look at the semantics to figure out which input should output an error in the form of an exception. Next, a small pivot program for each exception was written, that asserted, that the specific exception would be thrown. The small programs then each contained a syntax and or semantic error, that the compiler then must locate and throw the correct exception.

The following code, see code snippet 7.2 is a Pivot program, that contains an error, that the compiler must handle. In this example the code tries to compare an int with a float. This is not legal in Pivot.

```
init(){
    int a = 5;
    float b = 5.0;
    // Should throw DifferentTypesComparisonException, since 'a' is int
    and 'b' is a float
    if(a < b){
        print "Hej";
    }
}
```

Code snippet 7.2: Pivot program with comparison of different types

The code in code snippet 7.3 show the tests method for the pivot program in snippet 7.2.

```
@Test
void TestWrongReturnTypeException(){
    Compiler.setSourceFile("WrongReturnTypeException.pvt");
    assertThrows(WrongReturnTypeException.class, Compiler::compileToJava);
}
```

Code snippet 7.3: Testing that the program throws the correct exception

25 of these tests were written and accompanied by 25 small pivot programs with a corresponding error. Some errors were initially not found by the compiler, but these tests led to solutions of these errors. For this reason these tests improved the reliability of the Pivot compiler.

7.2 System tests

In order to make sure, that the compiler as a whole also works, some system tests were conducted.

The first system tests consisted of compiling some programs, that were known to have the correct syntax and well defined expressions. It was then asserted, that the compiler would accept these programs and generate a runnable program in the java language. Some of these programs were up to 200 lines of Pivot code, and every combination of events was inserted to make sure, that all were accepted by the compiler. Four of these major system tests were conducted, all of which passed. The tests were run in an almost identical way to the unit tests, see section 7.1.2, but simply with significantly bigger Pivot programs. The system tests also helped ensure, that the compiler behaved in a well defined and expected manner corresponding to the syntax and semantic rules.

7.2.1 Client server tests

The client server implementation is a tool that makes it possible for the written pivot code to communicate with the different devices to the server and back. This section of the program was made not only to make it easier for the user to implement their own home automation, but also to test if events that are triggered by signals, are handled correctly.

Once a pivot program is written and compiled, a server can be started while executing the pivot code. The server will search for devices that wishes to connect to the server. Once a device is connected, it is possible to send and receive signals to the devices from the server and back. When a signal triggers an event declared in the pivot code, the appropriate signal data is generated and sent to the device. The device should then read the signal data, switch it's state and send back a signal with its new state to the server. Multiple pivot programs were written and different device connections were tested, to make sure that everything works. The following tests were made:

- Do signals trigger the correct events?
- Ensuring that an already running event is terminated, when another of the same event is requested by the server
- Devices can connect and disconnect from the server without problems or crashes

Until now, the tests mentioned above are working correctly. The clients can disconnect and connect to the server again without creating duplicate devices. When an event is executed while another instance of the same event is running, the event will be reset and executed from the start again. All events are executed correctly and no bugs were found in the generated Java code. One problem, which we stumbled upon, is that when a device is disconnected from the server and reconnects, the device can connect from a new port. Optimally, devices should connect to the same port, since there are a limited amount of ports on a network. This would be something to work on in future works.

7.3 Concurrency testing

The Pivot program includes global variables as well as multi-threading. For this reason some run conditions can occur and must be handled properly, read more about the implementation of this in section 6.5.4. Concurrency can be difficult to test. One way to test this is to increment the same global variable inside two different concurrently running events. If both events for example increment the variable one thousand times, the variable should always end up being incremented two thousand times. The reason why this test, however, cannot definitely say, that the program has no run condition, is even if the concurrency is implemented in the wrong way, the result may end up being correct. If this is implemented wrongly, and the compiler still has a race condition, the result of the program depends on when the CPU is interrupted. This test can, however, show if something is definitely wrong, and for this reason it was conducted.

The code in code snippet 7.4 shows this test.

```
int i = 0;

init(){
}

every 10 hours{
    int k = 0;
    while(k < 1000){
        i = i + 1;
        print "thread 1: " + i;
        wait 10ms;
        k = k + 1;
    }
}

every 10 hours{
    int k = 0;
    while(k < 1000){
        i = i + 1;
        print "thread 2: " + i;
        wait 10ms;
        k = k + 1;
    }
}
```

Code snippet 7.4: Concurrency Pivot program for testing

This program prints to the screen the value of the variable *i* every time it is incremented. The value of *i* must then be two thousand after every execution. The test was run fifty times to ensure, that this was always the case, and all tests passed. However, this is still not a guarantee that no race condition can occur.

7.4 High level language test

The problem analysis showed a need for a higher level language that enables easier development of an event based program to control home automation. This led to the development of the Pivot programming language.

To make sure that Pivot is, in fact, a higher level language than the general purpose languages, which are oftenly used in IoT, a few analytic tests were made. The tests are based on the relation between the number of lines of pivot code and the number of lines of java code that is generated with the Pivot compiler. By comparing this data, it should be possible to find a tendency between the length of the pivot code and the length of the generated java code. However, it should be mentioned, that when the pivot code is compiled into generated java code, that java code may not optimally written for few lines of code. The test should, however, still provide reasonable results.

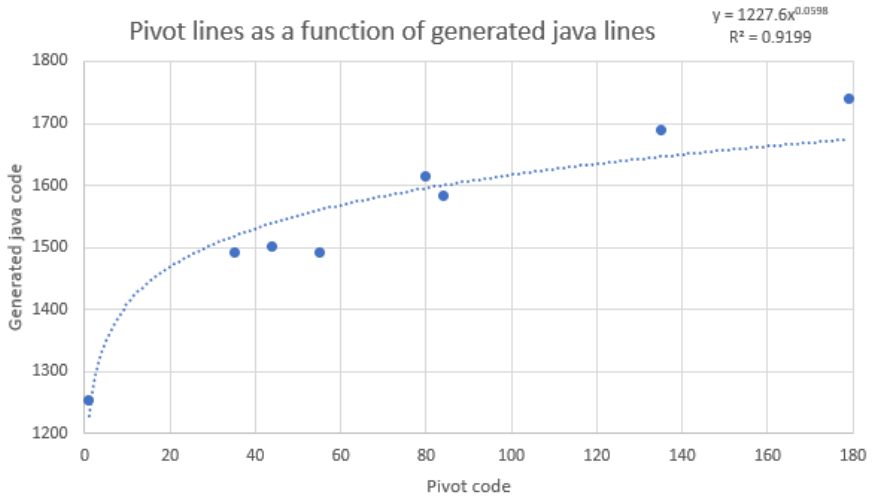

Figure 7.1: Shows a graph with the amount of pivot code lines as a function of generated java code lines.

As seen on the graph in figure 7.1, the tendency line shows the amount of generated java lines corresponding to the amount of pivot lines. The more pivot lines that are written, the less java lines per pivot line are needed to write the same code. The different data points in the graph represent written pivot programs that are compiled into generated java code. The graph intersects the x-axis at 1250, because it takes 1250 lines of boiler plate code to generate the server and generate all the standard abstract classes. As seen on the tendency line, it follows a power function, showing exactly what was initially the purpose with the Pivot language - to be able to create small home automation programs using our programming language, that would otherwise have been significantly bigger in a general purpose language.

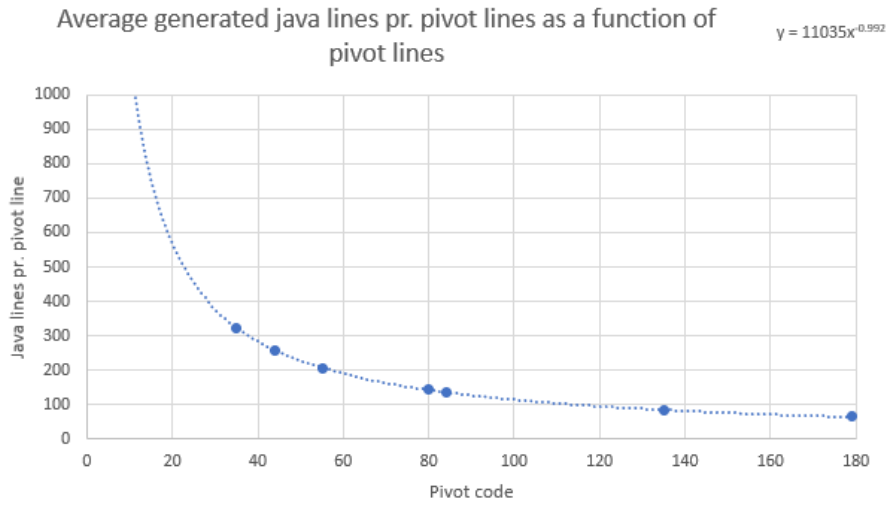

Figure 7.2: Shows a graph with the average amount of java lines needed pr. pivot lines

The graph in figure 7.2 shows, that as the pivot programs grow bigger, the less amount of java lines are needed to generate the line of pivot code. This means, that especially for smaller home automation programs many lines of code can be saved by using the Pivot language.

7.5 Usability test

Since reliability is the most important criteria, it is hard to test in a usability test. Readability and writability, however, can affect reliability, since it increases maintainability, see section 3.2.1. For this reason a readability test and writability test was conducted. This test was also an obvious chance to test, if the language was any good for the target user. This was ensured, since the target subject was a programmer with several years of programming experience.

The usability test was conducted with the subject sitting at the computer and controlling the mouse and keyboard. The IntelliJ IDE was used with word highlightning for the code.

The test was divided into two parts. First, one for readability, and then one for writability. This was done to enable to subject to get to know the syntax of the Pivot language before actually writing in it. For both tests, a test supervisor was present, that introduced the tasks and asked questions if needed.

The entire usability test can be found in appendix A.

7.5.1 Readability

The readability test was a task to see, if the subject was able to understand the behaviour of the program just by looking at the code. In order to quantify the usability test, some expected words were written down before the tests. The expected words were all synonyms for the exact keywords in the Pivot language. If the subject used any of these words, it was assumed, that he/she also understood the behaviour.

The readability was divided into five parts, where each part had some different code with different expected words, see appendix A.1.1 for all parts including code. The results are summarised in table 7.1.

	Passed	**Remarks**
Part 1	X	
Part 2	X	
Part 3	X	
Part 4	X	Misunderstood date
Part 5	X	

Table 7.1: Readability test summary

For part 1 the subject said type definition and type. Two of the expected words. For this reason it is assumed, that the subject understood the expected behaviour of the code snippet.

In part 2 the subject said the words initialisation as well as variable name and declaration. For this reason it can be assumed, that the subject understood that this code snippet contained variable declaration of devices.

In part 3 the code contained the init function. For this code section, the subject used the words block and start of program. The subject discussed, that this code appears to run before everything else in the program, which was also correct. For this reason it is assumed, that this was also understandable to the subject.

In part 4 the code contained two different events. In this part, the subject used many of the expected words. All the following words were mentioned: Event, time, date, start time, repeat, set function, sleep, delay, input, output and if statement. It appears, that the subject understood everything. Here, however, the subject was confused about the time format. He understood *01d04m* as 1 day and 4 minutes, but in Pivot this actually means the first of April.

In part 5 the subject also used many of the expected words, function call, return type, function id, argument, parameter type, parameter id, function declaration. For this reason, it is also assumed, that the subject understood the meaning of the

code.

It appears that for the most part, the syntax of the Pivot programming language fits the intended behaviour of the Pivot program according to the target user. This is assumed, since for all five parts, the subject used the correct synonyms for the code. It can, however, not be definitely concluded with only one test subject.

7.5.2 Writability

The writability test was conducted by giving the subject some tasks, that he/she should express a solution for in the Pivot language. The subject was allowed to look at the code from the previous readability test to gather inspiration. The test was considered passed, if the code could compile, and the supervisor accepted the code to solve the task at hand. The writability test was divided into two tasks, see appendix A.1.2. The first task asked the subject to write a program in Pivot, where a light bulb would turn on every 2 hours. The results of the writability tests are summarised in table 7.2.

	Passed	**Remarks**
Task 1	X	Misunderstood input/output
Task 2	X	

Table 7.2: Writability test summary

Initially, the subject had trouble understanding what input and output meant. In the Pivot language a device output means, that the device can output some signal to the server. The subject, however, expected the output from the servers point of view, meaning the other way around. This should be considered in future revisions. Other than this misunderstanding, the rest of task 1 was completely quickly and correctly.

The second task involved the subject making and addition to the program. Now a door should open, when a sensor sense a person at the door. This task was also completed fairly quickly and with little confusion. There was, however, one compile error, where a comma was mistaken for a semi colon, which resulted in a parsing error. The error message to the user was in this case not very useful to the subject.

It appears that a subject of the target audience would be able to express his/her target program in the Pivot language. The parser errors, however, should be modified.

7.5.3 Summary

In conclusion the Pivot language appears to be readable to the target audience. There were only minor misunderstandings in the readability test. The Pivot language also appears to be writable for a test subject from the target audience, but with one exception, that the *input*/*output* keywords may be misleading. Also the compile error exceptions, in the Pivot parser, were not very helpful. It can, however, not be definitely concluded with only a single test subject.

Chapter 8

Closure

This chapter will discuss the final outcomes of this project. This includes a discussion of the solution as well as both a conclusion and some future posibilities for the Pivot language.

8.1 Discussion

In this section, the final Pivot programming language will be evaluated. Additionally the development of the semantics and implementation of the programming language will be described. In section 3.3 it was decided that the most important criteria for the language, is to fulfil the reliability requirement. Furthermore, the target users are somewhat experienced programmers, which led to readability and writability being less important, but still worth investing effort into. To test the language criteria, a usability test was conducted. The purpose of this test was to test readability and writability, since it is hard to test reliability with a usability test. The test showed that Pivot, for the most part, fulfilled the design criteria, since the subject did not have any real problems reading or writing in Pivot.

Reliability has been tested by writing a lot of tests and creating error handling. Also a number of test programs were written and connected to a device simulator. Something that has not been tested very thoroughly however, is the generated code.

Pivot was initially designed to be a programming language for all devices. This was due to the fact that many of the current home automation products, like Philips Hue and Ikea Trådfri, allowed for some customisation, but did not offer the full control that comes with a programming language. This was fulfilled in a way that all devices can connect as long as they use the Pivot network protocol. Since the Pivot protocol consists only of strings it would not be hard to use with most devices.

One of the reasons to use Pivot is because it allows for more customisation than

solutions such as Philips Hue, but less than openHAB. This is because openHAB is a lower level language language than Pivot. Thus, openHAB does allow for more customisation than pivot, but potentially require more effort by the programmer. Additionally, openHAB already has driver support for many existing smart devices, which Pivot does not have, meaning that openHAB may currently be the better choice. In the future however, Pivot might provide a more simple, but still powerful experience, which could make it a better choice for novice programmers. Since Pivot is a higher level language than openHAB, one might prefer the syntax of Pivot, but this is up the personal preference of the user.

The behaviour of the Pivot language was described and formalised using operational semantics. While operational semantics allow for precise and formal definition of behaviour, it can also be difficult to read for some programmers. For this reason the semantics of the Pivot language could have also been written using regular English. This would have the advantage, that almost everyone would be able to read. However, it is difficult to get the exact behaviour described using English, which could lead to unexpected behaviour.

During the development of this project, many obstacles had to be overcome. The first was the abstract syntax, that had to be done early on due to one of the semester courses. The abstract syntax that was initially developed was very different to the one that Pivot ended up with. This means, that a couple of weeks was lost just figuring the abstract syntax out.

Additionally the idea of network connections in the Pivot language changed along the way. Initially, it was thought, that the devices would be initialised to some ip, that the device then had to use as a static ip on the network. This turned out to be a sub-optimal solution due to a lot of network management needed for it to work. Later it was then changed to be the device, that pinged the server using the Pivot network protocol.

Another major obstacle in the development of the Pivot language was the semantics. It was known and discussed quite early on the in process. The formal operational semantics were, however, unknown to every one in the project in the beginning. Also, the course that taught semantics only caught up in April. Since a lot of the implementation depended on the semantics, everything was delayed.

The network development part of the Pivot program also posed some problems. No member of the group had any experience in network programming, and the course for this is not until fifth semester at Computer Science bachelor at Aalborg University. For this reason a lot of research was needed, which further delayed the progress.

Concurrency of execution was also taught during the fourth semester. It was, however, not finished until late in April. This meant, that the concurrency implementation was not specified until a late point in the development. Also the semantics depended on the concurrency implementation.

Generally, many of the subjects needed for development of the Pivot language were unknown to everyone in the group in the beginning. This meant, that a lot of time was spent trying to learn new skills along the way. The members of the group learned a lot during this project, but it also meant, that many extra hours had to be put in to learn the new skills.

8.2 Conclusion

The Pivot language is an event based language, that can be compiled to Java using the Pivot compiler. The Pivot compiler is also written in Java, but with help from the ANTLR parser generator tool, see section 3.4 for target language analysis.

In section 2.3 some criteria for the language were set. The first being, that it must be easy to express and solve home automation problems. This requirement appears to be resolved based on the usability test, see section 7.5.

The second requirement said that the user must be able to control devices based on incoming signals and time based events. Incoming signals is accomplished by creating a central server, see section 6.5.4. Time based events can also be created in the language, see section 6.5.4 for implementation.

Since these features allow for parallel running of events, some concurrency implementation had to be done. This is solved using Java Threads, see section 6.5.4. Concurrency, however, introduces new problems with deadlocks. This problem was solved by having the locks for the global variables also be acquired in the same order, see semantics in section 5.2.4.

Another solution criteria was, that Pivot must be able to support addition of new types of devices. This problem was solved using the syntax seen in section 4.2. This syntax when generated to the target language, Java, creates Java classes, which represents the new devices in the running Pivot program.

In order to evaluate the solution, some language qualities was found in section 3.3. Reliability was the most important, which was achieved with locks, see section 5.2.4. Another way to increase reliability is to write tests. Both unit tests and system tests were conducted for the Pivot compiler, see section 7.2 and 7.1. Additionally, some concurrency testing had to be conducted to ensure, that the program behaves as formalised in the semantics, see section 7.3

Readability and writability was, however, also important, since they can also affect reliability. Readability and writability appears to be fitting to the target user according the usability test, see section 7.5. The usability test was, however, fairly

limited with only one test subject. The Pivot language also appears to be significantly higher level that for example Java, see section 7.4. This enables the user to write significantly bigger home automation programs with fewer lines of code.

Overall, the Pivot language solves most of the problems described in the problem description, with one fairly major exception. The Pivot programs cannot simply connect to all devices. However, a network protocol was designed, that all client devices are allowed to use in order to connect to the Pivot Server.

8.3 Future works

In this project, we were successful in making a good base for the pivot language. The essential functionality is working, and it is possible to actually use the language to write home automation programs to control various devices in a home. During the project initialisation stage, there were lots of ideas and functionality that we chose to ignore in the implementation, due to them not being essential. Some of these ideas were additions to the language and others were improvements to the current functionality.

8.3.1 Include

Include was an addition that we talked a lot about implementing. This functionality makes it possible for the user to split the code out on different files, as well as making libraries that can be used in various different programs. This would not only save time when writing pivot code, but also make it possible to structure the code a bit more. It was chosen to limit the project to only one file, because the pivot language is already a very high level language, that does not require lots of code to work, and to implement this feature would simply take too much time compared to the benefit that is gained.

8.3.2 Network connection

In the problem analysis, it was mentioned that there were requirements regarding how the server should communicate with the devices. As for now, the way communication is dealt with, is by having a simple protocol containing three different strings, describing which device to send to, what kind of signal type it is and the data the signal wants to inform the device. This is a very simple protocol, this does not apply to all kinds of devices. One solution is to write drivers, that translates our protocol to the appropriate protocol for the specific device we wish to send

data to and from. If we had more time it would be beneficial to find a real hardware device, such as a Philips hue bulb, write a driver for the lamp and use that for testing. For now, artificial devices that uses our current protocol was used for system tests as described in section 7.2.

8.3.3 Error handling

The usability test revealed some problems regarding error handling in pivot. The subject made it clear that it would be less time consuming to search for errors if the errors would be written directly in the console with the entire statement. Currently an error location and description is written in the console but not the code actually having the error. This is more time consuming to find the error since the user then has to switch between the console and the code to figure out the error. Another problem that was made clear was the lack of precision in parsing error. When a parsing error happens, the console writes "could not parse", the line number and reference to the line. A more effective thing would be to use the gcc compiler method and print an arrow that points to the illegal token. This expansion to error handling could decrease time spent on errors which would lead to an increase in a programmers efficiency.

8.3.4 Usability test

The usability test included in section 7.5 gave some good inside into how the target audience might think. It appears that the Pivot language fits the target user fairly well. The usability test, however, only included one test subject. In order to ensure, that it is the case, that Pivot fits the target audience, more usability tests should be conducted.

Bibliography

[1] antlr. *ANother Tool for Language Recognition*. https://www.antlr.org. 2019.

[2] Calaos. *Calaos official website*. https://calaos.fr/en/. 2019.

[3] Richard J. LeBLANC Jr. Charles N. Fischer Ron K. Cytron. *Crafing a Compiler*. Addison-Wesley Publishing Company , USA ©2009, 2009. ISBN: 0-13-606705-0, 978-0-13-606705-4.

[4] Bret Crawley. *Parser Generators: ANTLR vs JavaCC*. https://dzone.com/articles/antlr-and-javacc-parser-generators. 2016.

[5] Jacob Bach Hansen Søren Jensen Kenneth Toft Rasmussen Marc Kjær Deleuran Michael Tarp Dennis Vestergaard Værum Frederik Madsen Halberg. *Home Automation*. https://projekter.aau.dk/projekter/files/198203443/Report_Home_Automation.pdf. 2014.

[6] Thom Dietrich. *openHABian hassle-free openHAB Setup*. https://community.openhab.org/t/openhabian-hassle-free-openhab-setup/13379/670. 2017.

[7] Domoticz. *Domoticz official manual*. https://www.domoticz.com/DomoticzManual.pdf. 2019.

[8] GeeksForGeeks. *Java Naming Conventions*. https://www.geeksforgeeks.org/java-naming-conventions/. 2019.

[9] HomeAssistant. *HomeAssistant official website*. https://www.home-assistant.io/. 2019.

[10] Hans Hüttel. *Pilen ved træets rod*. Books on Demand, 2019.

[11] IBM. *How the JIT compiler optimizes code*. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_optimize.html. 2019.

[12] IBM. *The Internet Of Things (IOT) Will Be Massive In 2018*. https://www.forbes.com/sites/bernardmarr/2018/01/04/the-internet-of-things-iot-will-be-massive-in-2018-here-are-the-4-predictions-from-ibm/#22dad192edd3. 2018.

[13] Ikea. *Smart belysning*. https://www.ikea.com/dk/da/catalog/categories/departments/lighting/36812/. 2019.

[14] IoTforall. *Top 3 programming languages*. https://www.iotforall.com/2018-top-3-programming-languages-iot-development/. 2018.

[15] Jonathan Meyer and Daniel Reynaud. *Jasmin Home Page*. http://jasmin.sourceforge.net/. 2005.

[16] MisterHouse. *MisterHouse official website*. http://misterhouse.sourceforge.net/. 2019.

[17] Office for National Statistics. *Families and Households: 2017*. https://www.ons.gov.uk/peoplepopulationandcommunity/birthsdeathsandmarriages/families/bulletins/familiesandhouseholds/2017. 2017.

[18] OpenHAB. *OpenHAB official website*. https://www.openhab.org/. 2019.

[19] Oracle. *Avoiding Deadlock (Multithreaded Programming Guide)*. https://docs.oracle.com/cd/E19455-01/806-5257/6je9h0347/index.html. 2010.

[20] Oracle. *Java Thread Primitive Deprecated*. https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html. 2019.

[21] Philips. *Personal wireless lightning*. https://www2.meethue.com. 2019.

[22] ResearchGate. *DSLvsGPL*. https://www.researchgate.net/publication/228922345_Comparing_General-Purpose_and_Domain-Specific_Languages_An_Empirical_Study. 2010.

[23] safewise. *What is Home Automatiion*. https://www.safewise.com/home-security-faq/how-does-home-automation-work/. 2018.

[24] Robert W. Sebesta. *Concepts of Programming Languages*. 10th. Pearson, 2012. ISBN: 0273769103, 9780273769101.

[25] Michael Sipser. *Introduction to the theory of computation third edition*. Cengage Learning, USA ©2012, 2013. ISBN: 978-1-133-18779-0.

[26] Statista. *Number of Smart Homes forecast in the United Kingdom from 2017 to 2023 (in millions)*. https://www.statista.com/statistics/477134/number-of-smart-homes-in-the-smart-home-market-in-the-united-kingdom/. 2019.

Appendix A

Usability Test

A.1 Usability test questions

A.1.1 Readability test:

(Bold words were said by the subject during the usability test.)

Part 1:
What behaviour would you expect from the following code?
Expected words: **type definition**, **type**, class, enum, range, field, type inference

Part 2:
What behaviour would you expect from the following code?
Expected words: variable, object, **declaration**, **initialization**, **variable name**

Part 3:
What behaviour would you expect from the following code?
Expected words: scope, **block**, main, main function, print to terminal, string, **Start of program**

Part 4:
What behaviour would you expect from the following code?
What is the difference between when and every?
Expected words: **Event**, **time**, **date**, **start time**, april 1st 2019, statement, operator, **repeat**, **set function**, current time, **sleep**, **delay**, **input**, **output**, control structure, **if statement**, boolean

Part 5:
What behaviour would you expect from the following code?

Expected words: **function call**, **return type**, **functions id**, function definition, **argument**, input parameters, formal parameters, **parameter type**, **parameter id**, boolean, **function declaration**

A.1.2 Writability test

Task 1:
Write a program, where a bulb turns on every 2 hours.

Task 2:
Add to the same program:
Make a door open, when movement sensor outputs, that someone is at the door.

A.1.3 Readability code

```
// Part 1
define Signal toggle: on = 1, off = 0;
define Signal openWindow: 0.0..100.0;
define Signal celsius:  50..70;

define Device Bulb input: toggle;
define Device Window input: openWindow;
define Device LawnMower input: toggle;

define Device MovementSensor output: toggle;
define Device Thermometer output: celsius;


// Part 2
Bulb frontDoorLightOne = "frontDoorLightOne";
Bulb frontDoorLightTwo = "frontDoorLightTwo";
Window bedRoomWindow = "bedRoomWindow";
LawnMower GardenMower = "GardenMower";

Thermometer mainThermometer = "mainThermometer";
MovementSensor frontDoorSensor = "frontDoorSensor";


// Part 3
init(){
    print "Starting up...";
}

// Part 4
every 2 weeks starting 11:00 01d04m2019y{
    if(01d04m < now && now < 30d09m){
        set GardenMower toggle on;
```

```
33      }
34  }
35
36  when frontDoorSensor toggle on{
37      if(now > 17:00 && now < 23:00){
38          set frontDoorLightOne toggle on;
39          set frontDoorLightTwo toggle on;
40      }
41
42      wait 60 seconds;
43
44      set frontDoorLightOne toggle off;
45      set frontDoorLightTwo toggle off;
46  }
47
48  // Part 5
49  when 14:00{
50      if(get mainThermometer celsius > 25){
51          setBedroomWindowsOpen(100.0);
52      }
53  }
54
55  void setBedroomWindowsOpen(float percentage){
56      set bedRoomWindow openWindow percentage;
57  }
```

Code snippet A.1: Test code for readability test

Appendix B

Semantics

B.1 Boolean small-step transitions

[EQUALS-1]
$$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 == ex_2, sto > \Rightarrow < ex_1' == ex_2, sto >}$$

[EQUALS-2]
$$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 == ex_2, sto > \Rightarrow < n_1 == ex_2', sto >}$$

[EQUALS-3]
$$env_v \vdash < n_1 == n_2, sto > \Rightarrow_b tt \quad \text{if } n_1 = n_2$$

[EQUALS-4]
$$env_v \vdash < n_1 == n_2, sto > \Rightarrow_b ff \quad \text{if } n_1 \neq n_2$$

[SMALLER THAN-1]
$$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 < ex_2, sto > \Rightarrow < ex_1' < ex_2, sto >}$$

[SMALLER THAN-2]
$$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 < ex_2, sto > \Rightarrow < n_1 < ex_2', sto >}$$

[SMALLER THAN-3]
$$env_v \vdash < n_1 < n_2, sto > \Rightarrow_b tt \quad \text{if } n_1 < n_2$$

[SMALLER THAN-4] $$env_v \vdash < n_1 \texttt{<} n_2, sto > \Rightarrow_b ff \quad \text{if } n_1 \not< n_2$$

[BIGGER THAN-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{>} ex_2, sto > \Rightarrow < ex_1' \texttt{>} ex_2, sto >}$$

[BIGGER THAN-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{>} ex_2, sto > \Rightarrow < n_1 \texttt{>} ex_2', sto >}$$

[BIGGER THAN-3] $$env_v \vdash < n_1 \texttt{>} n_2, sto > \Rightarrow_b tt \quad \text{if } n_1 > n_2$$

[BIGGER THAN-4] $$env_v \vdash < n_1 \texttt{>} n_2, sto > \Rightarrow_b ff \quad \text{if } n_1 \not> n_2$$

[NOT-EQUAL-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{ != } ex_2, sto > \Rightarrow < ex_1' \texttt{ != } ex_2, sto >}$$

[NOT-EQUAL-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{ != } ex_2, sto > \Rightarrow < n_1 \texttt{ != } ex_2', sto >}$$

[NOT-EQUAL-3] $$env_v \vdash < n_1 \texttt{ != } n_2, sto > \Rightarrow_b tt \quad \text{If } n_1 \neq n_2$$

[NOT-EQUAL-4] $$env_v \vdash < n_1 \texttt{ != } n_2, sto > \Rightarrow_b ff \quad \text{If } n_1 = n_2$$

[GREATER-OR-EQUAL-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{>=} ex_2, sto > \Rightarrow < ex_1' \texttt{>=} ex_2, sto >}$$

[GREATER-OR-EQUAL-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{>=} ex_2, sto > \Rightarrow < n_1 \texttt{>=} ex_2', sto >}$$

[GREATER-OR-EQUAL-3] $$env_v \vdash < n_1 \texttt{>=} n_2, sto > \Rightarrow_b tt \quad \text{If } n_1 \geq n_2$$

[GREATER-OR-EQUAL-4] $env_v \vdash < n_1 \texttt{>=} n_2, sto > \Rightarrow_b ff$ If $n_1 \not\geq n_2$

[SMALLER-OR-EQUAL-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{<=} ex_2, sto > \Rightarrow < ex_1' \texttt{<=} ex_2, sto >}$$

[SMALLER-OR-EQUAL-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{<=} ex_2, sto > \Rightarrow < n_1 \texttt{<=} ex_2', sto >}$$

[SMALLER-OR-EQUAL-3] $env_v \vdash < n_1 \texttt{<=} n_2, sto > \Rightarrow_b tt$ If $n_1 \leq n_2$

[SMALLER-OR-EQUAL-4] $env_v \vdash < n_1 \texttt{<=} n_2, sto > \Rightarrow_b ff$ If $n_1 \not\leq n_2$

[PARENT-B] $$\frac{env_v \vdash < b, sto > \Rightarrow_b < b', sto >}{s \vdash < (b), sto > \Rightarrow_b < b', sto >}$$

[AND-1] $env_v \vdash < tt \texttt{ \&\& } tt, sto > \Rightarrow_b tt$

[AND-2] $env_v \vdash < b \texttt{ \&\& } ff, sto > \Rightarrow_b ff$

[AND-3] $env_v \vdash < ff \texttt{ \&\& } b, sto > \Rightarrow_b ff$

[AND-4] $$\frac{env_v \vdash < b_1, sto > \Rightarrow_b < b_1', sto >}{env_v \vdash < b_1 \texttt{ \&\& } b_2, sto > \Rightarrow < b_1' \texttt{ \&\& } b_2, sto >}$$

[AND-5] $$\frac{env_v \vdash < b_2, sto > \Rightarrow_b < b_2', sto >}{env_v \vdash < tt \texttt{ \&\& } b_2, sto > \Rightarrow_b < tt \texttt{ \&\& } b_2', sto >}$$

[AND-6] $$\frac{env_v \vdash < b_2, sto > \Rightarrow_b < b_2', sto >}{env_v \vdash < ff \texttt{ \&\& } b_2, sto > \Rightarrow_b < ff \texttt{ \&\& } b_2', sto >}$$

[AND-7]	$env_v \vdash < b_1 \ \&\& \ b_2, sto > \Rightarrow_b ff \quad$ If $b_1 \wedge b_2 \Rightarrow_b ff$
[AND-8]	$env_v \vdash < b_1 \ \&\& \ b_2, sto > \Rightarrow_b tt \quad$ If $b_1 \wedge b_2 \Rightarrow_b tt$
[OR-1]	$env_v \vdash < tt \ \| \| \ b, sto > \Rightarrow_b tt$
[OR-2]	$env_v \vdash < b \ \| \| \ tt, sto > \Rightarrow_b tt$
[OR-3]	$env_v \vdash < ff \ \| \| \ ff, sto > \Rightarrow_b ff$
[OR-4]	$\dfrac{env_v \vdash < b_1, sto > \Rightarrow_b < b_1', sto >}{env_v \vdash < b_1 \ \| \| \ b_2, sto > \Rightarrow < b_1' \ \| \| \ b_2, sto >}$
[OR-5]	$\dfrac{env_v \vdash < b_2, sto > \Rightarrow_b < b_2', sto >}{env_v \vdash < tt \ \| \| \ b_2, sto > \Rightarrow < tt \ \| \| \ b_2', sto >}$
[OR-6]	$\dfrac{env_v \vdash < b_2, sto > \Rightarrow_b < b_2', sto >}{env_v \vdash < ff \ \| \| \ b_2, sto > \Rightarrow < ff \ \| \| \ b_2', sto >}$
[OR-7]	$env_v \vdash < b_1 \ \| \| \ b_2, sto > \Rightarrow_b ff \quad$ If $b_1 \vee b_2 \Rightarrow_b ff$
[OR-8]	$env_v \vdash < b_1 \ \| \| \ b_2, sto > \Rightarrow_b tt \quad$ If $b_1 \vee b_2 \Rightarrow_b tt$

Table B.1: Small step semantics for boolean operations

B.2 Arithmetic small-step transitions

[PLUS-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{+} ex_2, sto > \Rightarrow < ex_1' \texttt{+} ex_2, sto >}$$

[PLUS-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{+} ex_2, sto > \Rightarrow < n_1 \texttt{+} n_2', sto >}$$

[PLUS-3] $$env_v \vdash < n_1 \texttt{+} n_2, sto > \Rightarrow n \qquad \text{where } n = n_1 + n_2$$

[SUB-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{-} ex_2, sto > \Rightarrow < ex_1' \texttt{-} ex_2, sto >}$$

[SUB-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{-} ex_2, sto > \Rightarrow < n_1 \texttt{-} ex_2', sto >}$$

[SUB-3] $$env_v \vdash < n_1 \texttt{-} n_2, sto > \Rightarrow n \qquad \text{if } \; n = n_1 - n_2$$

[MULT-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 * ex_2, sto > \Rightarrow < ex_1' * ex_2, sto >}$$

[MULT-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 * ex_2, sto > \Rightarrow < n_1 * ex_2', sto >}$$

[MULT-3] $$env_v \vdash < n_1 * n_2, sto > \Rightarrow n \qquad \text{if } n = n_1 \cdot n_2$$

[DIV-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < ex_1 \texttt{ / } ex_2, sto > \Rightarrow < ex_1' \texttt{ / } ex_2, sto >}$$

[DIV-2] $$\frac{env_v \vdash < ex_2, sto > \Rightarrow < ex_2', sto >}{env_v \vdash < n_1 \texttt{ / } ex_2, sto > \Rightarrow < n_1 \texttt{ / } ex_2', sto >}$$

[DIV-3] $$env_v \vdash < n_1 \texttt{ / } n_2, sto > \Rightarrow n \qquad \text{if } n = n_1 / n_2$$

[PARENT-1] $$\frac{env_v \vdash < ex_1, sto > \Rightarrow < ex_1', sto >}{env_v \vdash < (ex_1), sto > \Rightarrow < (ex_1'), sto >}$$

[PARENT-2] $$env_v \vdash < (n), sto > \Rightarrow n$$

[NUM] $$env_v \vdash < v, sto > \Rightarrow n \qquad \text{if } \mathbb{R}[\![v]\!] = n$$

[$\text{ADD}_{\text{STRING}}$] $$env_v \vdash < str_1 \texttt{+} str_2, sto > \Rightarrow str \qquad \text{if } str = str_1 \circ str_2$$

[STRING]	$\mathbf{L} = \{[a-z], [A-Z], [0-9], \; '\_', '-', '!', ' ', '.', ':', '+', '/', '='\}$
	$env_v \vdash < v, sto > \Rightarrow str \quad \text{if } \mathbf{L}[\![v]\!] = str$

Table B.2: Small step semantics for arithmetic operations