Dataflow language for implementation of artificial neural network models

Design, definition and implementation of programming languages

D402f19

Spring 2019
Project report P4

Aalborg Universitet
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst

AALBORG UNIVERSITY
STUDENT REPORT

House of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst
http://aau.dk

Title:
Dataflow language for implementation of Artificial Neural Network models

Project Period:
4. semester

Project Group:
D402f19

Participants:
Christian Graae Zandersen
Daniel Thomsen
Jesper Adriaan van Diepen
Mike Lund Andersen
Niki Ewald Zakariassen
Simon Steiner

Supervisor:
Florian Lorber

Page Numbers: 146
Standard pages: 69 (Excl. Appendices)

Date of Completion: 28/5 - 2019

Abstract:

The sudden growth of machine learning has resulted in a shortage of the required skill-set in the industry. The popular educational tools for machine learning are built on top of general-purpose languages and thereby restricted by its syntax. The aim of this project is to create a programming language with educational capabilities through simple syntax with a high level of readability and writability. The language is restricted in scope by limiting it to artificial neural network models. A dataflow approach is taken to increase the cohesion between the program structure and the mental model of the programmer. The language is limited in its expressivity as it is not Turing complete, but instead a purely structural language.

Table Of Contents

Preface

The source code for the compiler to the proposed language can be found in the accompanying files, or at the Github-page: `https://github.com/niki9796dk/P4-CompilerAndLanguage`. The generated Javadoc-page is also included as a zip-folder in the accompanying project files.

Aalborg University - May 28, 2019

Christian Graae Zandersen
<czande17@student.aau.dk>

Daniel Thomsen
<dthom16@student.aau.dk>

Jesper Adriaan van Diepen
<jvandi15@student.aau.dk>

Mike Lund Andersen
<mikand17@student.aau.dk>

Niki Ewald Zakariassen
<nzakar17@student.aau.dk>

Simon Steiner
<ssimon17@student.aau.dk>

1. Introduction

Machine learning has become more involved in large companies worldwide and will continue to grow in the coming years [1][2]. Different visualization methods for machine learning are used to explain the connectivity between individual elements, typically through directed graphs. Current visualization techniques do not represent current implementations in the most commonly used programming languages. [3]

Currently, the most used languages for machine learning are general-purpose programming languages, such as Python and R [4]. These languages have the option to include various machine learning libraries in order to increase the ease of development in machine learning. This can increased development quality but still has its limitations. Being general-purpose languages, they will include a multitude of unnecessary basic constructs of the language, which the current solutions are bound to.

Machine learning is used within a wide range of research areas [5], where the researchers working with machine learning already have a high understanding of the capabilities of machine learning and is already familiar with the most used libraries, such as TensorFlow and Keras. As it is becoming an increasingly more useful skill in the industry[6][7], the educational tools should follow. Education in machine learning is affected by the, at times, illogical difference between the visualization of a model, and the code that describes it. This might make it more difficult to fully understand the given machine learning model.

Based on the presented arguments, this project will go in depth with the possibility of creating an educational tool for machine learning. The report will go through all the phases of analyzing the aforementioned points by designing and developing a tool based on that analysis. The first part of the report will feature the problem analysis, where the problem and the relevant stakeholders will be identified, and a product will be proposed. Parts two and three will feature the design of the language and the translator. Part four of the report will feature the process of implementing a translator for the proposed language and ensure its quality. Lastly, the product will be reflected and concluded upon.

Part I
Problem Analysis

2. Societal Context

To solve a problem, one needs to understand the broader impact it has on its context, and thereby narrow it down to a few key points. This includes analyzing the initial problem as well as examining the stakeholders in the context.

2.1 Initial Problem

Working with machine learning (ML) is not an uncommon occupation in the software industry. According to a report by Econsultancy and Adobe, based on a "survey of 12,795 marketing, creative and technology professionals in the digital industry"[8], 15% of businesses used ML as of February 2018, and 31% said that it was on the agenda for the next 12 months [8]. It follows that organizations are interested in having tools available for their developers, to streamline the development of ML software. On top of that, interest grows around the area from researchers and enthusiasts.

Current programming languages for working with ML include R and Python. R was created for statistical computing and is influenced by multiple paradigms i.e. Imperative, Object-Oriented and Functional [9]. Python was created as a general-purpose language, that emphasizes code readability, and is also influenced by the Imperative, Object-Oriented and Functional paradigms [10]. Their popularity in machine learning is partly due to their large collection of relevant libraries [11]. This allows programmers to use libraries that other programmers have made available online.

Data Scientist at Intel, Will Koehrsen states that:

"The future of machine learning lies not in one-off solutions but in a general-purpose framework allowing data scientists to rapidly develop solutions for all the problems they face." [12]

This is the direction that R and Python libraries are moving towards. The downside of these frameworks being implemented as libraries in another language is that the libraries are restricted by the constructs, syntax, and semantics already implemented in this language. This might affect the programmer's ability to make a connection between the way they think about the problem and the way that the program ultimately ends up solving the problem.

Developers need the ability to efficiently be able to model the desired solution mentally, and afterward be able to implement it as it was modelled. This is what is being attempted in R and Python through their libraries, but due to the vastness and complexity of these languages, they might fall short. One might only be familiar with a subset of the language or its libraries, and thereby lose both readability and writability. It is argued that a programming language designed with a more concentrated and narrow scope would allow programmers to use the language constructs more efficiently and effectively, as the syntax and semantics would be made with the sole purpose of solving a specific area of problems and consequently work as an educational tool.

On the basis of the preceding analysis of the initial problem, the following points can define the key problems:

- Languages being used for machine learning were designed for too broad a range of applications
- Libraries are restricted by the predefined language constructs and syntax.
- Implementations using libraries in general-purpose languages won't accurately reflect the mental model of the programmer.

To further expand on these points and the general problem, an analysis of the stakeholders will be described in the following section.

2.2 Stakeholder Analysis

To determine which stakeholders exist within the context of the problem and what influence they have, it's important to identify and prioritize possible stakeholders. Furthermore, this section will also cover the advantages and disadvantages the stakeholders stand to face from the project.

As a result of the analysis; primary and secondary stakeholders will have been identified, which makes developing the proposed language in the context of these stakeholders possible.

2.2.1 Identifying Stakeholders

The stakeholders that will be affected by this project will have some level of knowledge within programming and specifically an interest in machine learning.

Most of the stakeholders can be segmented into two primary groups:

1. The potential users
2. The language and framework alternatives

The potential stakeholders in the first segment are:

- Researchers
- Engineers
- Teachers
- Students
- Enthusiasts

These are all groups who would be potential users of a programming language designed specifically for developing machine learning applications. The second segment comprises of the following stakeholders, among others:

- Python (Python Development Team)
    Tensorflow (Google Brain Team)
    Keras (Google Brain Team)
    NumPy (Community project)
    SciPy (Community library project)
- R (R Core team)

In theory, all programming languages and ML tools would be relevant stakeholders in this segment, but these are the primary ones. [13]

On top of the two segments, two additional self-contained stakeholders were identified:

- Developers of the intermediate compilation language that will be used in this project
- The project group itself

To further examine the role that the identified stakeholders play in relation to the initial problem at hand, they will be prioritized in the following subsection.

2.2.2 Prioritizing Stakeholders

Potential stakeholders can be prioritized based on two parameters, by their power and by their interest in the project. [14]

- Power is determined by how much a stakeholder is able to influence the project.
- Interest is determined by how interested the stakeholder is in the success or failure of the project.

This categorization is therefore easily visualized using a stakeholder matrix as in Figure 2.1, where the identified stakeholders have been mapped.

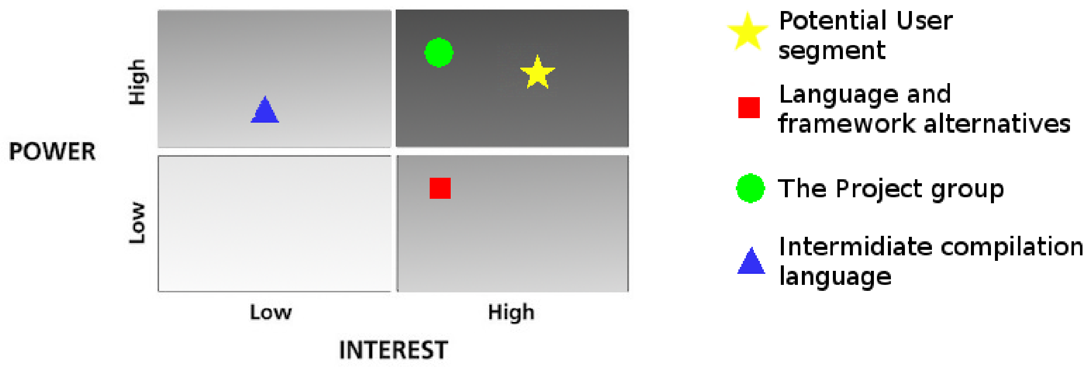

POWER
High
Low
Low
High
INTEREST
Potential User segment
Language and framework alternatives
The Project group
Intermidiate compilation language

Figure 2.1: An overview of the stakeholder matrix

The lower left quadrant represents stakeholders that do not influence, nor have a high interest in the project. Their relation to the project is secondary. None of the identified stakeholders reside in this quadrant.

High interest and low level of power in the lower right quadrant. This quadrant contains different machine learning language and framework alternatives. These will act as the opposing parties towards the language.

Due to the market share that these languages and libraries hold [15], they will be prioritized as a less important stakeholder on a general level.

Stakeholders located in the upper left quadrant have a high level of power and a low level of interest. As Figure 2.1 illustrates, this quadrant is occupied by the intermediate language that the project's language is going to compile to. Their power is indirect in the way that if they decide to change the syntax of their language, this affects the project's translator, which will most likely require an update. On a general level, they will be prioritized as less important as most high-level languages don't change or remove core functionality or syntax, they instead add more.

Finally, the upper right quadrant has a high level of power and high interest. This quadrant contains the stakeholders in the potential user segment and the project group. Since the project group is able to make decisions nearly independently of other stakeholders, and are directly affected by the success of the project. As a project group that studies computer science and learns about machine learning, the group becomes part of the user segment as well.

As for the user segment as a whole, they have a potential interest in the success of the project, and partly due to this, they will also have the power to influence the outcome of the project.
Due to both the interest and power of these stakeholders, they are potentially very important to the project and to help define its success.

To further examine the user segment, the individual stakeholders are prioritized.

Prioritizing stakeholders in the User segment

In the User segment, the following machine learning-involved stakeholders were included: Researchers, Engineers, Teachers, Students, and Enthusiasts.

It is reasonable to assert that companies and research teams would benefit from robust and efficient implementations of machine learning algorithms, as this would guarantee a certain level of certainty that their results are reproducible and reliable in a business context. As evidence of this, they often rely on large and well-tested libraries developed by a large team, as can be seen on the market share of each implementation of machine learning, where a few implementations have a disproportional market share.[15]

As a small team designing an entirely new programming language, it can be difficult to achieve the same level of depth and efficiency. Therefore, the focus will be put on the other half of the user segment, specifically the teacher, student, and enthusiast stakeholders.

As a student learning about machine learning algorithms, the most important part will often be about understanding the underlying model and dataflow [Appendix F]. It could be argued that simply applying a library to a problem, might not help a student to a better understanding of the underlying dataflow of a selected model. By allowing students to quickly implement simple algorithms, as they are modelled, without having to understand every single detail might help the students to an increased understanding.

To summarize, the potential stakeholders have been identified and prioritized. It has been discovered that the potential user segment will be the most relevant stakeholder for this project. Furthermore, the potential user segment has been narrowed down to the following group of users: Teachers, students, and enthusiasts in the context of machine learning. Moving forward, the primary focus will be on this user group.

2.2.3 Stakeholder Advantages

To further examine the user segment, their advantages will now be identified. To examine stakeholder advantages the following questions will be answered [14]:

1. What is the stakeholder's interest in the project? What is their motive and goal?
2. Which pros and cons the stakeholder will experience as a result of the project?
3. Which contributions could the stakeholder make towards the project, both positive and negative ones?

As it was shown in the stakeholder matrix 2.1, the user segment has a high interest in the project, as they are the target group, which the language is primarily made for. The language should serve as a good entry level language for people wanting to learn machine learning. A reasonable assumption is that these users will have some knowledge of programming beforehand. A possible goal for the potential users of the language could be that the syntax should be easily understandable, given existing knowledge of basic programming.

Machine learning teachers and students would experience both positives and negatives as a result of this project. A positive is that the language will be made for the ease of learning, to improve the understanding of machine learning models. The existing solutions available consist mainly of libraries, which makes it easy to implement machine learning algorithms, while making it harder for newcomers to understand how the algorithms actually work.
A negative could be that the solution will be a new language, which means that the teachers and students will have to learn new syntax, to be able to use the solution.

Both machine learning teachers, students, and enthusiasts could contribute to the development of the project. Teachers would contribute by sharing the main goals of a machine learning course, to make sure that the languages' core features would aid in fulfilling said goals. Machine learning students and enthusiasts would contribute to the project, by sharing their views on which features have the highest impact on writability, readability, and reliability, seen from a learning perspective.

2.2.4 Stakeholder Involvement

This part of the stakeholder analysis will discuss how the stakeholders should be treated during the development of the project, and how information will be received from them.

As mentioned in section 2.2.3, contributions to the project will be made by teachers teaching machine learning, and by the students machine learning students. Information will be gathered from a teacher through an interview. The interview method allows data gathering to be done in a semi-structured way, by having a set of broad and open-ended questions ready beforehand. The direction of the interview can be altered along the way, depending on the answers given, and thus, creating a dialogue over a specific topic. It is believed that this will be the best way to include machine learning teachers in the project.

The remaining group of stakeholders consists of machine learning students and enthusiasts. The information there is to gain from the informants, consists mostly of personal preferences i.e. which properties of a programming language are of most importance, and what properties are associates with the terms writability, readability, and reliability. Students and enthusiasts will contribute during the usability evaluation phase. Usability evaluation will be done in the later stage of development. In the later stage of the project a usability evaluation will be held, by asking fellow students to write small programs in the proposed language, and give feedback. This will be discussed further in Chapter 14

Summary

The central problem is the immense prevalence of people working with machine learning, while there is no dedicated language with a fitting syntax for the use case. Students, enthusiasts, and teachers as potential users will be prioritized, as creating a specialized language can help in reducing the learning curve for people inexperienced in machine learning. The next step is examining existing tools to identify points of possible improvement.

3. Existing Solutions

Although the existing languages and libraries were not prioritized as a general stakeholder, some of them might still hold power regarding the project. This power is expressed through its influence on the proposed language. The most influential machine learning tool at the time of writing is by far TensorFlow, looking at GitHub activity, google searches, books, articles and job listings [15]. The second most influential solution is Keras which is an API that sits on top of TensorFlow, which was made to make it easier to get into machine learning. Keras was also highlighted in the interview [F] as the solution to compare the proposed language to. To analyze the shortcomings of existing solutions and look at how the initial thoughts of the proposed language compare to them, Keras will be used as the underlying basis of comparison, since it in many ways has the same goal as the proposed language.

Keras builds on top of TensorFlow which is a library implemented in Python. As mentioned in the Initial Problem [Section 2.1], Python is a general-purpose language influenced by the Imperative, Object Oriented and Functional paradigms. Due to the nature of libraries, TensorFlow and Keras implicitly fall under these paradigms as well. Despite this, the more conceptual workings of these solutions are implemented Dataflow oriented, which is a central way of thinking about machine learning.

3.1 Dataflow programming

In software, dataflow is a paradigm that includes a range of approaches to thinking about program architectures that all share the feature of being data-centered [16]. Dataflow programming is a programming paradigm that internally represents programs as directed graphs, where the directed edges steer data between nodes [17]. Nodes have some data processing capabilities, which manipulate the incoming data and forward it through to its output. This representation of a program allows it to react to incoming data, without the programmer necessarily specifying how. Some of the immediate advantages to this way of thinking about programs is the level of abstraction that it allows the programmer. [16]

A simple way of visualizing how data flows through the directed graph is by drawing the directed graph as a diagram. It follows from this that a straightforward way of implementing dataflow in a programming language would be a purely visual programming language [17]. An example of how dataflow programming can be visualized as a graph is shown in Figure 3.1.

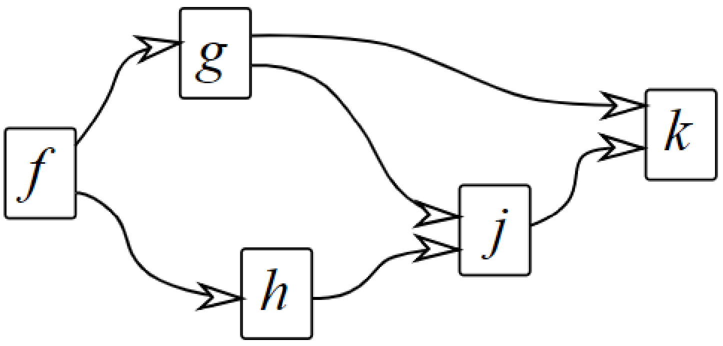


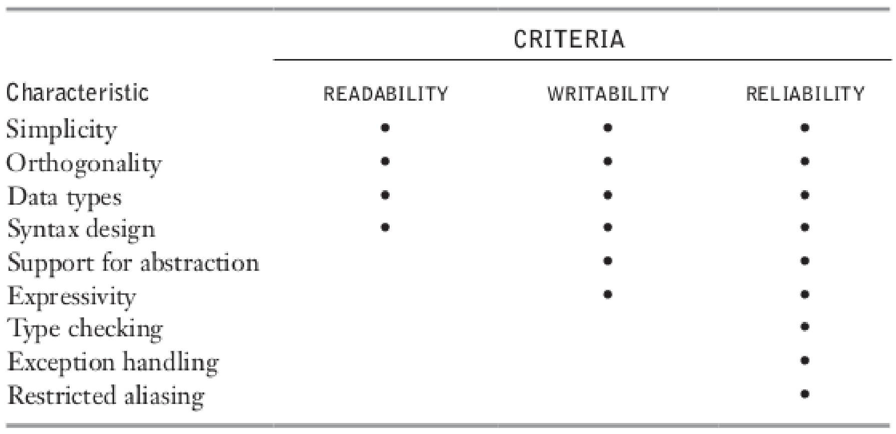
Figure 3.1: Dataflow programming visualized by a directed graph

3.2 Language Criteria comparison

To evaluate Keras as the existing educational solution in machine learning, and examine areas that could be improved, the Language Criteria model is used to make comparisons to propose improvements to these areas. An illustration of the central criteria for the evaluation of programming languages and the characteristics that influence them is shown in Figure 3.2.

Each of the overall criterion will be described, using the most relevant characteristics, and used to compare some of the key aspects of the proposed language with their Keras equivalents. While discussing the criteria, the project will focus on the examples in Listing 3.1 and 3.2, where each example consists of a Python part, using Keras, and an equivalent code snippet written in the proposed language.

Listing 3.1: Simple model in Python using Keras



	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 [18]

```
model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Listing 3.2: Simple model in proposed language

```
this.input
-> (build Layer([20, 64], Draw _Relu))
-> (build Layer([64, 64], Draw _Relu))
-> (build Layer([64, 10], Draw _Softmax))
-> this.output;
```

On top of this simple neural network, another syntactical comparison is made between Keras and the proposed language which highlights the restricting features of being a library, and not a standalone language. The models show how a custom layer will be implemented.

Listing 3.3: Making custom layer in Keras

```
from keras import backend as K
from keras.layers import Layer

class MyLayer(Layer):

def __init__(self, output_dim, **kwargs):
   self.output_dim = output_dim
   super(MyLayer, self).__init__(**kwargs)

```

```
def build(self, input_shape):
   self.kernel = self.add_weight(name='kernel',
                                 shape=(input_shape[1],
                                    self.output_dim),
                                 initializer='uniform',
                                 trainable=True)
   super(MyLayer, self).build(input_shape)

def call(self, x):
   return K.dot(x, self.kernel)

def compute_output_shape(self, input_shape):
   return (input_shape[0], self.output_dim)
```

Listing 3.4: Making custom layer in proposed language

```
block myLayer {
   mychannel:in input;
   mychannel:out output;

   blueprint(size weightSize) {
      source weights = build Source(weightSize);
      operation matrixMult = build Multiplication();

      (this.input, weights) -> matrixMult -> this.output;
   }
}
```

3.2.1 Readability

One of the core criterion for judging a programming language is readability. Readability is the ease with which programs can be read and understood [18]. Readability should be considered in the context of the problem domain, because a program written in a language unsuited for the specific use case, might be unnatural and difficult to read. There is an immediately noticeable difference in readability between the two languages shown above in Listings 3.1 and 3.2. As Keras is a library written in Python, it is bound by the basic constructs of Python, and by object-oriented programming. Python is a large general-purpose language, which means that there are many constructs that are unnecessary in the context of machine learning. The proposed language of this project will be designed with the sole purpose of being centered around machine learning, specifically ANNs, and it will be purely dataflow oriented. Therefore one could argue, that the readability of the proposed language is superior to that of Python, as it is more suited to the context of the problem domain. Some arguments to why the proposed language is more readable are as follows:

- *Simplicity:* A language is said to be simple if there is a relatively small amount of basic constructs in the language and if there are not multiple ways to accomplish a particular operation. Keras is made in a general-purpose language, and will therefore be bound by the constructs of Python, and naturally its simplicity will suffer from that in a machine learning context. The proposed language will be developed in such a way, that nothing will be implemented which is not useful in a machine learning context.
- *Orthogonality:* Orthogonality means that a relatively small amount of constructs can be combined in a relatively small number of ways [18]. Again it can be argued, that the proposed language will only have constructs that are relevant in a machine learning context, and therefore redundancies in the form of constructs will be minimized.

As mentioned earlier, readability is important when judging a programming language, and even more so, if said programming language will have its primary purpose in education. Therefore creating a language that enhances readability will have a high priority in this project.

3.2.2 Writability

Writability is a measure of how easily a program for a chosen problem domain can be created. Most language characteristics that affect readability also affect writability [18]. When comparing writability, it is important to consider the context of the problem domain. This makes it somewhat difficult to compare Keras, or Python, to the proposed language, as one of the languages is made specifically for the application in question, and the other is not. As mentioned when discussing readability, the proposed language will have constructs suited specifically for machine learning. Therefore many of the redundancies, present in Python in a machine learning context, are simply not present in the proposed language. On top of this, Keras is reliant on many Python constructs, but as shown in Listing 3.3 it is quite extensive to make a new custom layer class in Keras. Listing 3.4 illustrates nearly equivalent code in the proposed language which results in an ANN layer block which has trainable weights. The syntax of the proposed language is yet to be explained but will be in Chapter 7.

Another topic to consider, when discussing writability, is expressivity. Expressivity commonly means that a language is convenient, rather than cumbersome [18]. More specifically, it means that there are some very powerful operators, that allow a significant amount of computation to be done in individual, e.g. allows the programmer to express much in a short piece of code. In Listings 3.3 and 3.4 above, it is quite apparent that the expressivity of the proposed language is superior to that of Keras. Making custom layers in the proposed language is quite straightforward, as the math is already predefined in the language. As the proposed language will be created in the dataflow programming paradigm, mentioned in Section 3.1, one could argue, that the support for abstraction is better in the proposed language, as it's one of the main advantages of dataflow programming.

Like readability, writability will be considered of high priority in this project. The two criteria are closely related and they both define how programs in the language are developed and maintained.

3.2.3 Reliability

The last language criterion is reliability. A program is said to be reliable if it performs to its specifications under all conditions [18]. The first two criteria, readability and writability influence reliability. A key characteristic of reliability is exception handling. Exception handling is a program's ability to detect and intercept run-time errors. Big languages include extensive capabilities for exception handling. On this characteristic, the proposed language will differ significantly. The proposed language will be a smaller language intended for learning purposes, and not designed for huge research projects. For this reason, the proposed language is not safety critical, and the characteristics that affect reliability will be prioritized less, compared to those that affect readability and writability.

Summary

In this chapter, the proposed language was compared to the most closely related state-of-the-art solution, Keras, a library implemented in Python. The comparison was made using the language criteria; readability, writability, and reliability. It was found that the proposed language should aim to have better readability and reliability compared to Keras. This is because it will have a very specific use case, where Python, which Keras is dependent on, is a language made for a general-purpose and therefore needs several constructs not suited for machine learning. On top of this, a general focus should be put on readability and writability due to the educational purpose of the language. The last criterion, reliability, will be of lower priority, as it will not serve as much purpose in a learning context as the other criteria will.

4. Methodology

This chapter will focus on determining which methodology will be used to work within this project. The following methods each have scenarios in which they would be optimal to increase the quality of a project.

4.1 Development Approaches

There are two general approaches to handling project development strategies. These are the waterfall approach and the iterative approach. For some projects, a suitable middle ground can also be found through understanding the advantages and disadvantages of each. [19]

4.1.1 Waterfall Approach

The waterfall approach, also known as the traditional approach, consists of a linear sequence of activities. Each activity in the development process is being worked with individually.

This traditional approach often succeeds when the project development is well planned. The method does not allow for volatile changes to the project and is therefore inflexible. This may result in issues during the later stages of the development process. This is because decisions from early stages will affect later stages linearly. Some conditions might change throughout the development which causes the final product to degrade in quality. To avoid these scenarios, the activities are divided into three phases, all of which finish with overall quality assurance. If the quality assurance does not yield good results, the developers may reevaluate the contents of that current phase. [19]

4.1.2 Iterative Approach

The iterative approach handles the same activities as the waterfall approach. In contrast, the iterative approach works in a circular fashion.

This method will move through all activities quickly on the surface and then repeat that process through ***n*** iterations. Each iteration will then expand the depth of each activity phase. The final iteration will usually be determined by factors such as; time, budget or satisfaction.[19]

This method brings a significantly higher level of flexibility to the project, as prior discoveries and decisions are constantly reevaluated. [20]

4.2 Choice of Approach

The guidelines which the group has to follow throughout this project has to be taken into account in order to choose the best approach. This means there is a fixed final deadline, and it is expected that the group will present the language design portion of the project at the halfway mark, which functions as a soft intermediate deadline. Many of the necessary tools and concepts for the project are included in lectures that the group attends across the first 3/4 of the project, because of this keeping the project on pace with these can be advantageous.

Based on this, the project development strategy which will be used is a hybrid approach between the waterfall and the iterative approach, where the ability to iterate back and reevaluate prior phases is retained. The goal of the first core iteration is to produce a minimum viable product for the project. After this, additional shorter iterations will be done to revise choices that turned out unfavorable later in the process. As well as add functionality that was deemed valuable but not critical.

Summary

The project will be developed with a hybrid approach between the iterative and waterfall approaches, due to the project following the courses with a timeline identical to the lectures, with the possibility of adding more iterations after a minimal viable product has been developed. With a project development cycle strategy selected, the next step is to decide what type of system will be developed for this project.

5. Choice of Product

This chapter will focus on the choice of the product that should be developed in this project.

To limit the project to a specific area, going forward, the main focus will be on intuitive programming of machine learning models, and specifically the creation of artificial neural networks algorithms, as they are easily modelable. Furthermore, it has been decided to develop a functioning compiler as the translator choice, because the context of the language may consist of computationally heavy elements. This choice also allows the use of an intermediate translation language which can both be an interpreter and compiler.

5.1 Problem Definition

To specify the exact definition of the problem, which will determine the agenda for the rest of the project, a summary of prior discoveries will be evaluated. Based on that evaluation, a problem statement and several sub-problems will be defined.

Currently, most language options that can be used for machine learning are general-purpose languages. This means that the user might be overwhelmed by the vast amount of constructs that the languages offer when they have to do a specific task within a subset of the language's use cases.

Those general-purpose languages are good for different kinds of libraries, but using libraries developed for a specific language, results in the user having to use a specific syntax for that language to use the library. The syntax in the library is therefore not reflective of its specific capabilities.

The stakeholder analysis identifies the relevant stakeholders and concludes that the most central stakeholder groups are students, teachers, and enthusiasts which entails that the proposed language should primarily focus on its educational purposes.

The analysis of the problem area uncovered several similar tools and languages currently being used. In the Existing Solutions [Chapter 3], Keras in specific is further analyzed and compared to the proposed language based on the overall language criteria to find areas in which improvements might be made.

Based on these discoveries the following problem statement will be examined:

How can a programming language be designed and its appurtenant compiler implemented, to help increase the educational potential, of working with Artificial Neural Networks?

To answer this general statement, it's needed to examine and answer these sub-statements:

- Which syntactic, contextual and semantic language constructs will allow a high level of readability and writability.
- Which information should the individual compiler phases accept and output to allow translation from the proposed language to Java code that can be further compiled into Java byte-code and machine code - and how should such phases be implemented.
- How can the educational potential and language criteria be evaluated?

As described in Section 4.2, this project will be developed following the iterative approach. This affects how the problem definition will be answered quite significantly. The first iteration will be the longest one and will result in a minimum viable product, and as such, will have answered the problem statement and sub-statements at a reasonable level.

With new knowledge about the problem at hand and the direction that the project is to take, the requirements needed to solve it will be examined in the following chapter.

6. Requirement Specification

This chapter will address the requirements surrounding the proposed language, which should be fulfilled throughout this project. It is important that all requirements are testable for future evaluation purposes and compilation of the language.

On the basis of the involvement analysis in the stakeholder analysis from Section 2.2.4, an interview with a university instructor on the elective fifth-semester machine intelligence course on the Computer Science degree on AAU was conducted. Some of the specific points and suggestions that were presented in the interview will be presented and analyzed in these sections and outlined as requirements.

6.1 Interview

This section will highlight some of the key points and requirements that were gathered during the interview, and examine how they might affect the specific language and compiler requirements. The transcription of the interview is found in Appendix F. During the interview, the initial thoughts and ideas for the potential solution were described to the interviewee, where his thoughts on these are what will be used as the underlying basis for this section. The initial thoughts surrounding the language were described in Chapter 3 comparing the existing solutions to the proposed language.

- **I1: Gate renamed to Channel**
  The interviewee expressed concern about the term 'gate' because other machine learning models use the gate-term as something other than an entrance to a level of abstraction. The interviewee thought it would be better to rename it to 'channel' which would avoid a bias towards its meaning from users having worked with these other models. Channel refers to the general idea of a communications path over which information is moved.

- **I2: Connection notation as arrow**
  The notation to connect computational units in the language was received positively by the interviewee, who supported the '->' notation which was used in the proposed language.
- **I3: Strict sizes**
  In the proposed language, sizes were optionally given as parameters. The interviewee expressed concern towards this choice, as he believed it to be important in an educational environment that the programmers have to strictly describe the size of their input, and thereby explicitly be told if the sizes are not what you expected. Therefore, it should be required to specify.
- **I4: Mathematical details should be abstracted away**
  One of the features of the proposed language is the ability to do simple mathematical operations to process data. These operations are made available in the language, but users won't be able to make their own operations. This also entails that the mathematical details were hidden from users. The interviewee expressed his agreement towards this design choice.
- **I5: Know the target group**
  As mentioned previously, the interviewee is a university instructor on the elective fifth-semester machine intelligence course. Based on his experiences with this, he pointed out that based on the level of his students and the broadness of the curriculum, it wouldn't be fit for use as an educational tool on this semester. Instead, he mentioned that it would be more suited for postgraduate students who might specifically be interested in learning and modeling artificial neural networks.

The following sections will primarily focus on identifying requirements, specifically for the language and compiler, and prioritizing them in a table with the MoSCoW method. This will be done based on the interview transcribed in Appendix F, the problem analysis, and considerations made by the project group.

6.2 Language requirements

6.2.1 General language design

- **L1: Tokens defined with regular expressions**
  All tokens must be defined as regular expressions in order to recognize sequences in the lexical analysis of the compiler.

- **L2: Specified with context-free grammar in Backus-Naur Form**
  When specified as a context-free language, the language can be generated and verified according to this grammar.
- **L3: Unambiguous syntax**
  The syntax needs to be structured in such a way that different parsing methods will produce the same abstract syntax tree, again to avoid conflicting compiler implementations.
- **L4: Static and dynamic semantics**
  All semantics must be defined informally. The most central parts must also be defined formally using Structured Operational Semantics, to avoid any unclarity.
  The central parts will be identified in Chapter 11 when looking at the contextual constraints.
- **L5: The structure of the code should be representative of each element in dataflow diagrams**
  One of the stated goals of the proposed language is ease of use by making the translation from models to code more straightforward. To this end, each element of a dataflow diagram needs to have a direct parallel in the language, and be handled as components with specified connections between them.
- **L6: No useless non-terminals**
  The purpose of a grammar is to produce/verify syntactically valid strings of terminals in a language. Non-terminals, that in no way can result in terminals, are *useless*, and will only serve to clutter up the specification.
- **L7: Intuitive Language Comprehension**
  When the source code of similar neural network programs is compared to ones developed in the proposed language, a majority of users must judge the other language as less comprehensible.
- **L8: Highly Modifiable Language**
  The language must allow easy changes to the code and neural network structure, without requiring extensive development time.
- **L9: Self-descriptive Keywords**
  The keywords within the language must be appropriate for each keywords responsibility and correct usage context.
- **L10: Optional Indenting**
  The scope of a statement is defined by the use of brackets. Whitespaces, indents, and line-shifts are all ignored from the context.

6.2.2 Domain specific

- **L11: Custom operations**
  The programming language will consist of a predefined set of operations, but these operations might not cover all operations that the user would want to implement in a network. Therefore custom operations would serve as a way for the user to handle operations themselves.
- **L12: Recurrent blocks**
  The possibility of a connection between a block's output and its own input would be useful for creating recurrence within the networks.
- **L13: Nestable block structure**
  The option to create blocks within other blocks and create connections between them makes a block hierarchy easier to achieve.
- **L14: Predefined backpropagation and feed-forward for operations**
  A predefined set of operations should be included within the language. Because the main use of the programming language will be focusing on neural networks. It is natural to implement backpropagation and feed-forward for the operations, which allows the programmer to train the network.
- **L15: Fully and partly connected blocks**
  The language should be able to connect blocks fully, partly, and not at all. Although if any objects exist isolated within the code a warning should be thrown at compile-time. If any block has unused input channels, an error should be thrown, since dataflow programming wouldn't allow further progress in the dataflow in that scenario.
- **L16: Channels Connecting Blocks**
  The language must allow creating connections between blocks easily and without reducing readability.
- **L17: Block Definitions**
  The language must allow defining new blocks blueprints to easily allow repeated use of a similar structure.
- **L18: Block Procedures**
  The language must allow defining local code procedures which take a range of inputs. This should allow increasing the readability of code, as well as repeated use of the same procedures within the block.
- **L19: Limited Access to simple data-types**
  The language should focus on macro-programming of structuring

neural networks, and limit micro-programming such as direct data handling. Because of this, traditional data types such as Integers are not accessible.

- **L20: Procedure calling other procedures**
  The language should allow calling procedures within other procedures, to further increase code modifiability and readability.

6.3 Compiler requirements

- **C1: Adhere to the semantics.** The semantics of a language describes how it is to be interpreted. If the compiler doesn't follow them all, it can produce wrong outputs.
- **C2: Give useful error messages** When an error is detected at any point of compilation, the user needs to be informed of: That there is an error, wherein the code the error was detected, what step of compilation the error happened in, and a description of what might cause the given type of error.
- **C3: Graphical modelling of networks; Compiler Option**
  As requirement L5 lays out, the programming language will be designed to be implemented as dataflow diagrams are visually represented. By making it possible to generate a graphical model of the network, the user would be able to confirm their implementation.

6.4 MoSCoW

This section will create an overview of the importance of the requirements. Each requirement will be classified as either a must-, should-, could- or won't have feature. These classifications will be represented in a list, which shows the feature and the prioritization of that feature for the project as a whole.

This project follows a partially iterative approach, where the first iteration covers the requirements that are ***must have***. The option for further iteration will aim to implement the **should have**, and weigh the ***could have*** requirements against each other.

The following list highlights the 'must have' requirements, and the associated reasoning behind this choice.

- **L1 - Tokens defined with regular expressions:** Needed to make a Scanner and define how tokens are evaluated.

- **L2 - Specified with context-free grammar in Backus-Naur Form:** Needed to make the Parser and define which programs are legal.
- **L3 - Unambiguous Syntax:** Needed to make the parser.
- **L4 - Define static and dynamic semantics** Needed to make the symbol table and define which scoping and types are allowed and to ensure consistent behavior regardless of compiler implementation.
- **L5 - Structure of code representative of each element in dataflow diagram:** The core ideal of the language; allowing for greater ease of implementing neural networks.
- **L14 - Predefined backpropagation and feedforward for operations:** Without this, the language would need to accommodate the user defining operation functionality on their own, which would significantly expand the scope of the language, and needlessly burden the users. Also contributes greatly to the fulfillment of requirement L5.
- **C1 - Adhere to the semantics:** The semantics are defined to ensure consistency between compilers. It wouldn't do to not follow them.
- **L17 - Block Definitions:** Optimizing writability and reusability of code is essential to reduce development time for the user.
- **I4 - Mathematical details should be abstracted away:** Keeping the language structure and design-oriented instead of being heavy on mathematics makes the language more approachable and comprehensible. This is the key features for the language.
- **L8 - Highly Modifiable Language:** Machine learning is rarely perfect at first and requires small adjustments in order to acquire better results. Therefore making the language highly modifiable will greatly affect the language in a positive way.
- **L16 - Channels Connecting Blocks:** There has to be some interaction between blocks in order for the network to have any meaning or actually work correctly.

The other requirements are classified as ***should have***, ***could have*** or ***won't have***. The classifications and reasoning for these can be found in Appendix C.

Summary

By conducting an interview and using the problem analysis, a number of requirements were identified and split into the categories: Interview, language requirements, and compiler requirements. These were all prioritized using the MoSCoW model to find the 'must haves', and thereby move forward in the development knowing what to work towards. Having fully analyzed the problem and identified important requirements, the design of the language will now be discussed.

Part II

Language Design

7. Language Overview

In this chapter, the general language-constructs in the proposed language will be described. A code snippet was shown in Chapter 3 as an early comparison to the Python library Keras. As should be clear by now the purpose of the language is to allow easy structuring of an artificial neural network model. A potential model could be illustrated like in Figure 7.1, which illustrates an artificial neural network, and in continuation hereof also Figure 7.2 which illustrates how a layer might look.

The overall components when building these models is the general level of abstraction which are the grey boxes known as blocks. To allow reuse of blocks they are declared as a general blueprint which can later be built multiple times with specific parameters. An ANN layer such as the one in Figure 7.2 is a central example of a blueprint, and why it might be important when building a network with multiple similar layers. Another key component is a channel, which is what connects the blocks to each other allowing data to flow through them.

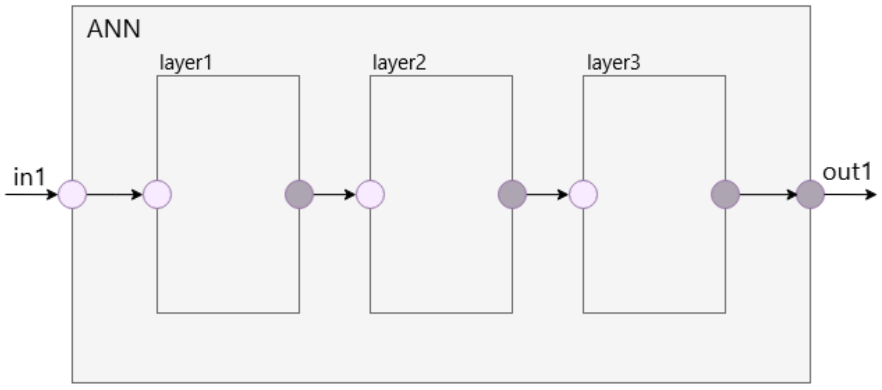


Figure 7.1: Illustration of an overall ANN block, depiction of Listing 7.1

Listing 7.1: ANN block described in proposed language

```
1  block ANN {
2     mychannel:in in1;
3     mychannel:out out1;
4
5     blueprint() {
6        block layer1 = build AnnLayer();
7        block layer2 = build AnnLayer();
8        block layer3 = build AnnLayer();
9
10       this.in1 -> layer1 -> layer2 -> layer3 -> this.out1;
11       this.in1 -> build AnnLayer() -> build AnnLayer() ->
            build AnnLayer() -> this.out1;
12    }
13 }
```

Listing 7.1 shows how one might express Figure 7.1 as a program in the proposed language. The first part of the block is declaring which 'in' and 'out' channels it has and specifies their identifier on lines 2 and 3. Then the blueprint for the block is declared on line 5, which consists of the contents of this block. In this case, it builds the AnnLayer block three times by looking at its blueprint, and assigns them to individual block-variables, on lines 6, 7, and 8. On line 10 the ANN block's in-channel is connected to layer1, which in turn is connected to layer2, connected to layer3 connected to the ANN block's out-channel. Due to the connection-notation being the lowest precedence element in the language, another option would be to build the AnnLayer blocks inline as is done on line 11. These methods of connecting blocks in a chain is only allowed when the middle elements have exactly one input-channel and one output-channel. If this was not the case, the connection would have to be split into multiple lines to specify which channels should be connected to which.

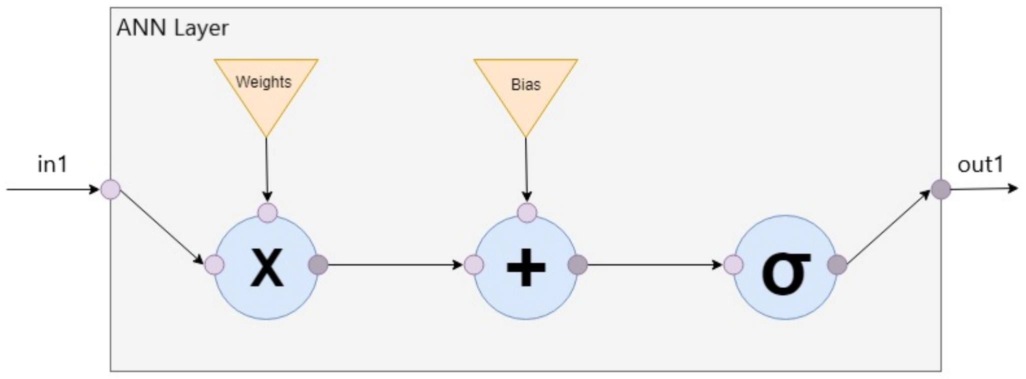


Figure 7.2: Illustration of a layer block in an ANN, depiction of Listing 7.2

Listing 7.2: ANN layer block described in proposed language

```
block AnnLayer {
    mychannel:in in1;
    mychannel:out out1;

    blueprint() {
        source weights = build Source();
        source bias = build Source();

        operation multiplication = build Multiplication();
        (this.in1, weights) -> multiplication

        operation addition = build _Addition();
        (multiplication, bias) -> addition

        operation sigmoid = build _Sigmoid();

        addition -> sigmoid -> this.out1;
    }
}
```

Listing 7.2 shows how one might express Figure 7.2 as a program in the proposed language. As previously, the channels are declared first, then the blueprint. This time the blueprint contains a few different constructs, such as source and operation. A source is a specific type of block that only has an output. An operation is also a specific type of block which is the only way of manipulating data, and could consequently be categorized as the smallest possible block. As described in Section 6.4 the requirement of facilitating custom operations was de-emphasized, and they are thus only available as predefined language-constructs. The underscore in front of the Addition and Sigmoid operations describe that they are unit-wise operations. The connection made on line 10 multiplies the input data with some weights whose result is added to the bias on line 13, which is connected to the unary operation: Sigmoid. The parentheses in the connections are called group connections and are especially convenient in regards to binary operations, as it connects each of the two blocks in the parentheses to the right operand of the arrow operator.

A few additional less-pivotal language constructs have been left out to keep this chapter general. These include the size type which can specify what data format is expected, procedures to allow another level of abstraction inside the blueprints, and use of parameters to parse data and generalize some blocks even more.

Summary

A description of general language constructs in the proposed language was made along with some examples of programs highlighting the proposed syntax. The chapter displayed how the proposed language allows easy modeling of graphs that data can flow through. A more detailed and in-depth specification of the syntax and semantics of the proposed language will be given on the basis of this overview.

8. Language Specification

To develop a programming language, it is important to define how the different elements of the language interact with each other. This chapter will focus on the syntax, contextual constraints, and semantics of the proposed programming language. Theory on context-free grammars is introduced to form a basic description of the syntax in the proposed language.

8.1 Context-Free Grammar

A Context-Free Grammar (CFG) is a language generator for describing the syntax of a language. It consists of a set of terminal and non-terminal symbols. For each non-terminal, there are one or more associated production rules. These production rules define how a non-terminal can be unfolded into an equivalent string of terminals and/or non-terminals. Terminals are tokens that carry an implication in the language. [21]

The language specified by a CFG is the set of possible sequences of terminals that can result from unfolding from the grammar's designated starting non-terminal. [21]

8.1.1 Backus-Naur Form

Backus-Naur Form (BNF) is a notation system for writing CFGs. In BNF, each non-terminal in the syntax is lined up with two colons and an equals sign (::=) or a right arrow ($\rightarrow$) followed by its associated rules separated with vertical lines (|). [21]

Extended Backus-Naur Form

Extended Backus-Naur Form (EBNF) are modifications of the basic BNF that allows for more compact grammar by introducing notation for common patterns like optional elements and repetitions. There are various variants of EBNF, because of this a specification of what extensions are used is necessary before use. [21]

The context-free grammar of this project will from here on be written in plain BNF, to take advantage of tools that only support the plain form.

8.2 Syntax

The syntax of a programming language specifies what can be written, understood and compiled. It determines the rules and processes of that language. The syntax comprises the lexical elements of the language. Lexemes, tokens, and terminals will all be defined and structured through non-terminals and the corresponding set of production rules.

8.2.1 Tokens

Lexical tokenization is the process of converting a sequence of characters into a sequence of tokens. Lexical tokens are recognized using regular expressions. A token has a defining regular expression which recognizes all possible token instances for each of the tokens. Table 8.1 defines the language tokens that should be ignored and their associated regular expression. These include newlines, white spaces, and comments.

White Space	Regular expression
newLine	$\backslash r \mid \backslash n \mid \backslash r\backslash n \mid \backslash n\backslash r$
whiteSpace	$newLine \mid [\ \backslash t\backslash f]$
comment	$oneLineComment \mid multiLineComment$
oneLineComment	$"//"[\hat{}\backslash r\backslash n]^* \ newLine$
multiLineComment	$[/][*][\hat{}*]^*[*]+([\hat{}*/][\hat{}*]^*[*]+)^*[/]$

Table 8.1: Table showing white space tokens and associated regular expressions

Table 8.2 shows the general tokens that have a multitude of different possible token instances.

Token	Regular expression
id	$[a-zA-Z\_]\ [a-zA-Z0-9\_]^*$
num	$[0-9]^+\ (\backslash.\ [0-9]^+)?$
sizeliteral	$\backslash[whiteSpace^*\ num\ whiteSpace^*, whiteSpace^*\ num\ whiteSpace^*\backslash]$

Table 8.2: Table showing general tokens and associated regular expressions

The keywords and symbols recognized in the token stream are defined in Appendix A, as the instance of these tokens is the same as the token itself.

As tokens are now recognized, the grammar can use them to check if they appear in syntactically legal configurations.

8.2.2 Abstract syntax

To specify a grammar, it is often easier to define the abstract syntax of the language and use that as a stepping stone to define the full set of production rules, that is parsable. The abstract syntax will describe the overall structure of the language. To do this, it assumes a collection of syntactic categories and each of them has a finite set of formation rules. The formation rules broadly define how the syntactic categories can be structured into a valid program [22]. The abstract syntax for the proposed language is defined in Table 8.3.

With abstract syntax defined, it's possible to go into detail with how the non-terminals and terminals should be structured in the production rules.

8.2.3 Production Rules

Figure 8.1 shows a small snippet of the full production rules which can be found in Appendix B. These production rules are used to specify the parsable syntax of the language in Backus-Naur form, based on the underlying structure of the abstract syntax.

The starting non-terminal is <*Prog*>, which defines a program to be a series of block definitions defined by <*Blocks*>.

As seen in the block definition, <*Definition*>, the overall structure of a block is simple and straightforward. First, the programmer must define a set of channels, after which he must define the blueprint and finally a set of procedures may be defined optionally. This order is strict and cannot be interchanged, which is determined to further increase readability. This is enforced by requiring that each part is where they are expected to be, otherwise, it won't parse.

The <*ChannelDeclarations*> part, while it might not be obvious from the snippet, requires at least one 'out' channel and an optional amount of 'in' channels.
To enforce this, the grammar splits the <*ChannelDeclarations*> into three subparts. The first subpart <*InChannelDeclaration*> specifies any optional

x	$\in$ Var – Variables
n	$\in$ Num – Numerals
b	$\in$ Bnames – Block names
p	$\in$ Pnames – Procedure names
e	$\in$ Exp – Expressions
C_{Chain}	$\in$ ConChain – Chain Connections
C_{Group}	$\in$ ConGroup – Group Connections
S	$\in$ Stm – Statements
D_C	$\in$ DecC – Channel declarations
D_V	$\in$ DecV – Variable declaration
D_{Bloc}	$\in$ DecBloc – Block declarations
D_P	$\in$ DecP – Procedure declarations
D_{Blue}	$\in$ DecBlue – Blueprint declarations
T	$\in$ Type – Types
D_{Bloc}	::= block b { D_C D_{Blue} D_P }
D_C	::= ε | D_C ; D_C | channel:in x | channel:out x
D_{Blue}	::= blueprint (T $x_1, ...,$ T x_k) { S } where $k \geq 0$
S	::= ε | S ; S | e
e	::= C_{Chain} | C_{Group} | D_V | x | p ($x_1, ...,$ x_k)
D_V	::= blueprint x = draw b | block x = build b ($x_1, ..., x_k$) | source x = build b ($x_1, ..., x_k$) | operation x = build b ($x_1, ..., x_k$) | size x = [n , n] | x = draw b | x = build b ($x_1, ..., x_k$) | x = [n , n] | D_V ; D_V | ε where $k \geq 0$
C_{Chain}	::= x_1 -> x_2 | x_1 -> ... -> x_k where $k \geq 0$
C_{Group}	::= (x_1, x_2) -> x_3 | (x_1, x_2) -> ... -> x_k
D_P	::= ε | D_P D_P | procedure p (T $x_1, ...,$ T x_k) { S } where $k \geq 0$
T	::= block | blueprint | source | channel:in | channel:out | size

Table 8.3: Abstract syntax – Syntactic categories and formation rules

```
# Program
<Prog>
   → <Blocks>

# Block
<Blocks>
   → <Block>
   | <Block> <Blocks>

<Block>
 → block id lcurly <Definition> rcurly

<Definition>
 → <ChannelDeclarations> <Blueprint> <Procedures>

# Channels
<ChannelDeclarations>
 → <InChannelDeclaration> <OutChannelDeclaration>
    <InOutChannelDeclaration>

# Blueprint
<Blueprint>
 → blueprint lpar <BlueParams> rpar lcurly <Content> rcurly

<Procedures>
 → procedure lpar <P_Params> rpar lcurly <Content> rcurly <Procedures>
 |  λ
```

Figure 8.1: A short snippet from the set of production rules, seen in Appendix B.

amount of 'in' channels, including zero, followed by the <*OutChannelDeclaration*> subpart, which specifies a single required 'out' channel. And finally, the last subpart specifies an optional amount of 'in' or 'out' channels. This allows the programmer to specify channels in any arbitrary order, while still enforcing at least a single 'out' channel to be defined.

Lastly, <*Blueprint*> and <*Procedures*> are defined in a very similar way. The only differences being that the programmer must define a single blueprint, but an optional number of procedures, and that the blueprint and procedure's parameters are restricted in different ways. The blueprint can only use parameters of the types: *blueprint* and *size*, while the procedure can use any type as a parameter.
The reasoning behind this distinction is to avoid allowing the programmer to create illegal connections between blocks.

An illegal connection is defined as a connection from one block to another, which does not go through the block's predefined 'in' and 'out' channels, but instead connect through the "wall" of a block, which is visually represented at Figure 8.2.
If the syntax allowed parsing blocks that have already been built as pa-

rameters to a blueprint, the programmer could then connect the internal blocks of that blueprint, to the channels of the external block which was parsed as a parameter, which is syntactically incorrect.

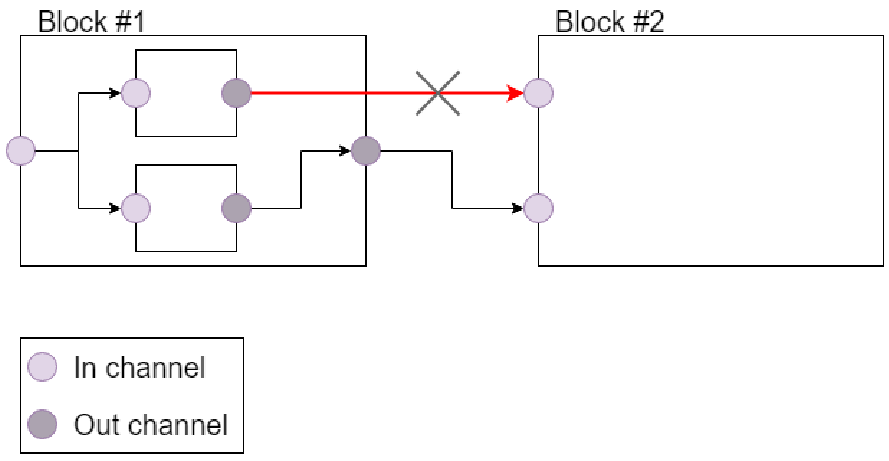


Figure 8.2: Visual representation of an illegal connection, marked as a red connection and a cross

8.2.4 Precedence and associativity

Precedence and associativity are issues that come up when an expression with two or more operators, is to be evaluated. This subsection will discuss the two terms one by one, and discuss how they will be used in the proposed language.

Precedence

Operator precedence is a relevant issue in an expression which include two or more different operators. An example of this is the following expression: $A + B * C$. With operator precedence the question is, which one of the two operators has the highest precedence and is therefore evaluated first, $+$ or $*$? This decision is made by the language designer. Typically arithmetic operators follow the same precedence rules in programming languages as they do mathematically, i.e. multiplication and division has higher precedence than addition and subtraction. [18]

Associativity

Associativity of operators is a relevant issue in expressions with two or more operators with the same level of precedence. As an example, the following expression is considered: $A * B/C$. With operator associativity there are two ways to evaluate such an expression, left to right, or right to left. Typically arithmetic operators are evaluated from left to right, i.e. $(A * B)/C$. [18]

An example of an operator which might be evaluated from right to left, is the assignment operator (=). Consider the following example: $A = B = C$. Under the assumption that the assignment operator is right-associative, the above expression will be evaluated as follows: $A = (B = C)$.

Precedence and associativity in the proposed language

As mentioned above, operator precedence and associativity is chosen by the language designer. The most central operators in the proposed language are: (), duild, draw, ., =, and ->, which are separated into five levels of precedence. Table 8.4 shows the rules of operator precedence and associativity for the proposed language.

Precedence level	Operator	Associativity
4	()	Left to right
3	build draw	Left to right
2	.	Left to right
1	=	Left to right
0	->	Left to right

Table 8.4: Precedence and associativity of the proposed language

As the table shows, the arrow operator (connection) has the lowest level of precedence, which means that connections are always evaluated last in any expression in the proposed language. To highlight precedence in the proposed language, the following expression is considered:

```
this.input -> block a = build Block(size b = [2,3]) -> this.output;
```

The example above shows that dot selection and assignments can be done between connections. In the proposed language, assignments are left-associative. The reason for this, is that chain-assignments are not legal in the proposed language.

With knowledge about the overall definition and specifications of the proposed language, it is possible to design the contextual constraints of that language.

8.3 Contextual constraints

Contextual constraints define the static semantics of the proposed language. The static semantics define the scoping and type system of the proposed language [21]. And as required by Requirement L4, all static semantics will be defined informally, and the most important parts formally.

8.3.1 Scope Rules

The scope of some name binding, such as a variable or procedure, refers to the region in which that binding is 'visible' [23]. Scope rules determine which bindings are valid during execution. Two kinds of binding can be used to determine scope rules: Static and dynamic scope rules. Static scope rules apply the bindings that were known when the named variable or procedure was defined, in contrast to dynamic scope rules which apply the bindings known when the named variable or procedure is encountered during execution. [22]

Most current programming languages use static scope rules due to the naturalness of reading and understanding the program. When writing programs it is difficult to read the code and know which parts bind how, as it will only bind during run-time [22]. The specific scope rules for the proposed language will be defined in the following sections. In accordance with the specified language requirement L4 regarding static and dynamic semantics, scoping will be defined both informally and formally.

Informal definition

In Section 3.2 the importance of readability and writability was highlighted. Due to this, and the wide use of static scoping in other popular languages, static scoping is chosen for the proposed language.

A program in the proposed language has a root scope in which a number of blocks can be declared. Each block has a scope in which channels can be declared, a blueprint is declared and procedures may be declared. A blueprint and each procedure have separate scopes as well, which are the deepest level of scopes that can be made. Both blueprint and procedures use parameters. Their formal parameters are treated as local variables within the respective scope.

Due to the strict structure and flat scope of the language, the static scoping is only relevant in relation to mychannels, which are declared first inside the block scope, but can later be changed within the blueprint or a procedure. This is illustrated in Listing 8.1 where the 'output' on line 8 refers to that of block b on line 7, and the 'output' on line 15 that the output channel of block c connects to refers to the one on line 3 in the current block. Given dynamic scope rules, the output in procA would instead refer to the one on line 7 as that is the current one where 'procA' is called instead of where it is declared. The code example is purely for demonstrational purposes and does not make sense practically.

Listing 8.1: Demonstration of when bindings are relevant

```
1   block blockA {
2      mychannel:in input;
3      mychannel:out output;
4
5      blueprint() {
6         block b = build blockB();
7         channel:out output = b.output;
8         b -> output;
9
10        procA();
11     }
12
13     procedure procA() {
14        block c = build blockC();
15        c.output -> output;
16     }
17  }
```

The use of the "this" keyword allows using the same id for multiple different declarations. It would be allowed to specify both a local channel and a block with the name of "a". To access the block one would simply write "a", and to access the local channel; "this.a". When specifying "this", the scope checker first looks for channels in the channel declaration scope, then individual procedure scopes, and lastly in its own local blueprint scope. Procedures are private to the block in which they have been declared, which is why local procedure variables won't be accessible to other scopes.

Environment-store-model

The environment-store-model is used to describe how variables, procedures and blocks bind. Variables bind to storage addresses, and each storage address holds a value, which consequently is the value of the variable. The general depiction of the environment-store-model is shown in Figure 8.3.

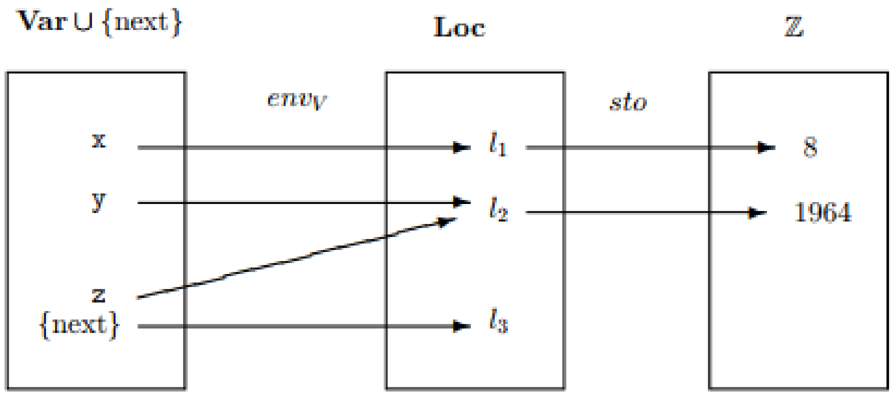


Figure 8.3: Depiction of the environment-store-model

Formal definition

The theory of the structural operational semantics is based on Transitions and Trees [22]. There are two different ways of describing an operational semantic, one is with big-step, and another is with small-step. In big-step semantics the transition $\gamma \to \gamma'$ describes an entire computation which starts in configuration γ and ends in configuration γ'. In small-step on the other hand, the transition $\gamma \to \gamma'$ describes a smaller step as part of the full computation. The small-step semantics can be more difficult to describe but is necessary for languages with parallelism. Due to the increased difficulty and lack of parallelism in the proposed language, big-step semantics will be used to formalize the language.

Equation 8.1 through 8.5 define the sets in the relevant environment store model. Equation 8.3 defines the procedure environment which is a partial function from the set of procedure names to statements and variables, where statements are its contents, and variables are the parameters and channels. Equation 8.4 defines the block environment as a partial function from block names to channel declarations, blueprint declarations and procedure declarations which all are its contents. Lastly, Equation 8.5 defines the blueprint environment which is equivalent to the procedure environment.

$$\mathbf{EnvV} = \mathbf{Var} \cup \{next\} \rightharpoonup \mathbf{Loc} \tag{8.1}$$
$$\mathbf{Sto} = \mathbf{Loc} \rightharpoonup \mathbb{Z} \tag{8.2}$$

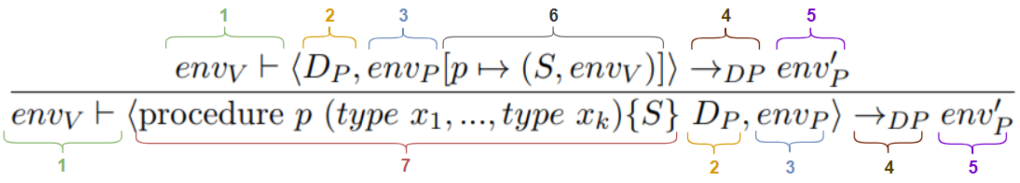
$$\mathbf{EnvP} = \mathbf{Pnames} \rightharpoonup \mathbf{Stm} \times \mathbf{Var} \tag{8.3}$$
$$\mathbf{EnvBloc} = \mathbf{Bnames} \rightharpoonup \mathbf{DecC} \times \mathbf{DecBlue} \times \mathbf{DecP} \tag{8.4}$$
$$\mathbf{EnvBlue} = \mathbf{Bnames} \rightharpoonup \mathbf{Stm} \times \mathbf{Var} \tag{8.5}$$

Equations 8.6 and 8.7 define rules for declaration of procedures with parameters under the assumption of fully static scope rules.

$$[Proc] \quad \frac{env_V \vdash \langle D_P, env_P[p \mapsto (S, env_V)]\rangle \rightarrow_{DP} env'_P}{env_V \vdash \langle \text{procedure } p\ (type\ x_1, ..., type\ x_k)\{S\}\ D_P, env_P\rangle \rightarrow_{DP} env'_P} \tag{8.6}$$

$$[Proc - Empty] \quad env_V \vdash \langle \epsilon, env_P\rangle \rightarrow_{DP} env_P \tag{8.7}$$

Equation 8.6 will be picked apart to show how these transition rules are built and what they describe. The bottom part of the equation is the conclusion, and the top part is the premise. A description of each part of the transition rule is illustrated in 8.4.



$$\frac{\overbrace{env_V \vdash}^{1} \langle \overbrace{D_P}^{2}, \overbrace{env_P}^{3}\overbrace{[p \mapsto (S, env_V)]}^{6}\rangle \overbrace{\rightarrow_{DP}}^{4} \overbrace{env'_P}^{5}}{\underbrace{env_V \vdash}_{1} \underbrace{\langle \text{procedure } p\ (type\ x_1, ..., type\ x_k)\{S\}}_{7}\ \underbrace{D_P}_{2}, \underbrace{env_P}_{3}\rangle \underbrace{\rightarrow_{DP}}_{4} \underbrace{env'_P}_{5}}$$

1. The variable-environment that the procedure sees
2. The procedure-declaration that follows procedure p
3. The procedure-environment before the procedure-declaration
4. The transition rules used to make the transition are the procedure-declaration rules
5. The procedure-environment after the procedure-declaration
6. The update of the procedure-environment
7. The procedure as written by the programmer, with identifier p, parameters with types and statements as its content

Figure 8.4: Description of components in transition rule 8.6

Equation 8.8 defines a rule for calling procedures following the fully static scope rules in procedure declarations. It is important to note that the variable environment given in the conclusion is not the same as the one in the premise because of the scope rules. The variables that were present when a procedure is called are not necessarily the same as the ones that were present when it was declared.

$$[Call - Proc] \quad \frac{env'_V[next \mapsto l] \vdash \langle S,\ sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle p\ (x_1, ..., x_k),\ sto\rangle \rightarrow sto'} \tag{8.8}$$

where $env_P p = (S, env'_V)$

and $l = env_V\ next$

Equation 8.9 defines the rule for declaring a block.

$$[Bloc] \quad \frac{env_V \vdash \langle D_{Bloc}, env_{Bloc}[b \mapsto (D_C, D_{Blue}, D_P)]\rangle \rightarrow env'_{Bloc}}{env_V \vdash \langle \text{block } b\ \{D_C\ D_{Blue}\ D_P\} D_{Bloc},\ env_{Bloc}\rangle} \tag{8.9}$$

Equation 8.10 defines the rule for instantiation of a block by building it.

$$[Var_{Decl_{Block}}] \quad \frac{\langle D_{V_{Bloc}}, env_V[x_a \mapsto l][next \mapsto new\ l], sto[l \mapsto v]\rangle \rightarrow (env'_V, sto')}{\langle \text{block } x_a = \text{build } x_b(x_1, ..., x_k)\ ;\ D_{V_{Bloc}}, env_V, sto\rangle \rightarrow (env'_V, sto')} \tag{8.10}$$

where $env_V, sto \vdash x_b \rightarrow v$

and $l = env_V next$

With scope rules covered, this chapter will continue with the type rules.

8.3.2 Type Rules

This subsection will describe the type system of the proposed language. Type systems are useful for avoiding typical logical errors, which might first arise suspicion at run-time, if at all.

An example of a type error which might go unnoticed in the dynamic language PHP, when using type-hinting, can be seen in Listing 8.2. In this example, PHP will implicitly try to convert the string into an integer, but instead of failing, since the value is not a number, it simply extracts the first characters, and convert those to an integer without throwing an error or warning, which might not be the expected behavior.

Listing 8.2: Type mismatch which might go unnoticed in the dynamic language PHP

```
function caller() {
   callee("25A27"); // echo 25
}

function callee(int $var) {
   echo $var;
}
```

Statically or dynamically typed language

Generally, there are two types of type systems, statically and dynamically typed languages. The difference between a statically typed and dynamically typed language is that static languages require the developer to specify variable types, which is not the case in a dynamically typed language. An example of the difference can be seen in Listing 8.3 and Listing 8.4.

Listing 8.3: Statically typed language example

```
int numberI = 25;
float numberF = 12.45;
boolean isStatic = true;
```

Listing 8.4: Dynamically typed language example

```
var numberI = 25;
var numberF = 12.45;
var isStatic = false;
```

The proposed language will be a mix of statically and dynamically typed. The reasoning behind building a mix can be traced back to Subsection 3.2.2, where the criterion of writability is declared of high priority. To achieve higher writability, the proposed language will from the perspective of the programmer, mostly be a dynamically typed language with only a few primitive types which need to be declared.
But from the perspective of the compiler, it'll function like a statically typed language, which will allow static type checking at compile-time.

While it might seem unintuitive how this mix of static and dynamic type system can be achieved, it is simply done by grouping a larger set of types into a single supertype, later referred to as sub- and supertypes.

An example of a supertype is the type *"block"*, which is defined as the full collection of user-defined blocks within the program. The user might have defined a block *ABC*, and when the programmer wants to build said block, he does not have to define a variable of the subtype *ABC*, but instead of the supertype *block*. This would be done as: block abc = build ABC(), as opposed to ABC abc = build ABC().
Therefore when the programmer needs to declare a variable, they simply refer to the set of supertypes instead of the individual subtypes, therefore making the language more dynamically typed, from the perspective of the programmer.

It is possible to know the subtype of a variable at all times, since the proposed language is not Turing Complete, hence no branching or iterations takes place. This lack of branching and iterations guarantee that statements are always executed, and always in the order, in which they were specified. This trait allows for simple tracking of a variables current subtype, depending on which line it was referenced. Because of this, static type-checking can be performed on otherwise dynamically typed code.

What is affected by type checking

Type checking systems are not a constant definition, as a result there are several variations between type systems from one language to another.

In the proposed language, the type system will affect the commonly used type constructs: Parameters, operators, variable declaration, and by extension: Reassignments of variables and build declarations of blueprint variables. The remaining semi-type relevant semantics, such as checking if a group connection matches the input size of the given type, will be checked in the semantic analysis part of the checking phases.

All type system checks will be checked against the supertype of the given variables/declarations since these were declared by the programmer. The subtypes will be used in the semantic analysis.

8.3.3 Formal type rules

In this subsection, the type rules for the main type-construct: Chains, will be defined. The type system for chains is defined by three type categories: M, I, O, the definition of which can be seen below.

$M ::= Block \mid Operation$

$I ::= M \mid Channel : in \mid Source$

$O ::= M \mid Channel : out$

The *M* group category is defined as the middle element group. The element group contains elements which are guaranteed to have both in- and out channels, and therefore can be connected to and from.
The *I* group category is defined as the input element group. The input element group contains elements which only guarantee to have out channels and therefore can only safely be connected from.
The *O* group category is defined as the output element group. The output element group contains elements which only guarantee to have in channels, and therefore can only safely be connected to.

Because of this distinction between *M*, *I* and *O* elements a simple informal rule for the chain type system says that:

All elements before the first connection must be *I* elements, as these are safe to connect from, while the last element in a chain must be an *O* element since these elements are safe to connect to. And everything in the middle must be an *M* element as these elements are the only ones to guarantee both in an out channels.

Equation 8.11 shows the type-rule for the simplest form of a connection which connects only two elements. The rule ensures that the first element is of type I, and the second of type O. Equations 8.12 through 8.14 shows the rest of the type-rules for the different ways of making connections between elements.

$$[C_{Chain_{Short}}] \quad \frac{E \vdash x_1 : I \quad E \vdash x_2 : O}{E \vdash x_1 -> \ x_2 : OK} \tag{8.11}$$

$$[C_{Chain_{Long}}] \quad \frac{E \vdash x_1 : I \quad E \vdash x_2 : M \ ... \ E \vdash x_{n-1} : M \quad E \vdash x_n : O}{E \vdash x_1 -> \ ... -> \ x_n : OK} \tag{8.12}$$

$$[C_{Group_{Short}}] \quad \frac{E \vdash x_1 : I \ ... \ E \vdash x_{n-1} : I \quad E \vdash x_n : O}{E \vdash (x_1, \ ... \ , x_{n-1}) -> \ x_n : OK} \qquad where, \ 2 < n \tag{8.13}$$

$$[C_{Group_{Long}}] \quad \frac{E \vdash x_1 : I \ ... \ E \vdash x_{n-1} : I \quad E \vdash x_n : M \ ... \ E \vdash x_{m-1} : M \quad E \vdash x_m : O}{E \vdash (x_1, \ ... \ , x_{n-1}) -> \ x_n -> \ ... -> \ x_m : OK} \qquad where, \ 2 < n < m \tag{8.14}$$

That concludes the type-rules, which is followed by a section describing semantics.

8.4 Semantics

The semantics of the language are quite traditional and straightforward. This is related to the limited uses of the language, and desire for ease of use for students with limited programming experience.

The language does not allow defining new subscopes within subscopes. The only subscopes allowed is a single blueprint and an optional amount of procedures, which are subscopes of a block. This means that code is limited to exist within a block's blueprint and procedures. This means that labeled references must be accessed within the current scope.

When connecting blocks using a chained connection statement, all blocks must be unary. This means they must only have a single input channel and a single output channel. The exception to this is if a parenthesis is used, which allows two or more blocks with a single output to be connected to a block with an equal amount of inputs. This is referred to as a group connection.

Chained channel connections allow both references to blocks, construction of blocks, as well as the initiation of blocks. When creating new blocks, the newly created block will be the one used in that part of the chain. Because of this, input and output channel counts must remain compatible.

Recursion is not allowed, and the rule is in the semantics of the language. An example of this is a blueprint containing a build instruction that directly or indirectly refers back to the blueprint with the original build call.

All outputs must be connected to an input channel in the blueprints of a block. This is to ensure all data channels flow directly from the original input to the final output, which is essential for both feed-forward and backpropagation to work correctly.

Summary

This chapter served as a specification of the proposed language constructs. The syntax was described using an abstract syntax, as well as the more detailed and unambiguous production rules which allow a deeper understanding of which constructs can be used wherein a program. Furthermore, a formal definition of both the scope and type rules was given, using structural operational semantics, which is a very concise way of describing something that would be difficult to describe informally and vulnerable to ambiguity. Lastly, the semantics of the language was described on a general level. That description involves how blocks are allowed to connect via channels and which blocks are allowed to be built inside other blocks. After describing the language, the translator allowing the use of such a language will now be designed.

Part III

Translator Design

9. Compiler Phases

A compiler has the task of translating a source language, to a target language. The target language can be any language, even a language that might still require additional steps until it can finally run on the hardware. These steps are often further compilation, such as from C to machine code. It can also be requiring a virtual machine to run within, such as the Java Virtual Machine.

9.1 Syntax Analysis

As was described in Section 8.2, syntax analysis is concerned with recognizing legal sequences of lexical elements.

9.1.1 Scanning

The scanner phase of the compiler, also known as the lexer phase, creates tokens from the source code by evaluating the regular expressions and categorizing each of the lexemes into the correct token classes. These tokens are then put into a token stream, which will later be used by the parser.

9.1.2 Parsing

The parser analyzes the token stream supplied by the scanner. By analyzing the order of tokens using the unambiguous production rules, it can be verified if the source code, is accepted by the grammar. If the source code is accepted by the grammar, a parse tree can be generated, and later an abstract syntax tree and finally a decorated abstract syntax tree.

9.1.3 Choice of tools

For this project, different tools will be used to build the scanner and the parser for the compiler. To build the scanner, the JFlex tool will be used, since JFlex is a scanner generator for Java, which takes a set of regular expressions and corresponding actions as input, and generates a lexer. The lexer will then generate the tokens, which will be used by the parser. JFlex is designed to work together with the parser generator CUP [24], and thus CUP has been chosen as the parser generator for this project.

The CUP tool generates LALR(1) parsers which use a bottom-up parsing approach. A parser generated by CUP can generate a parse tree while parsing, which can later be traversed to create the abstract syntax tree (AST). Instead of doing this, it was decided to skip the parse tree entirely and extend the parser with *Action Codes*, to build the AST directly from the production rules during parsing in the syntax analysis phase, which will be described in further detail in the implementation part.

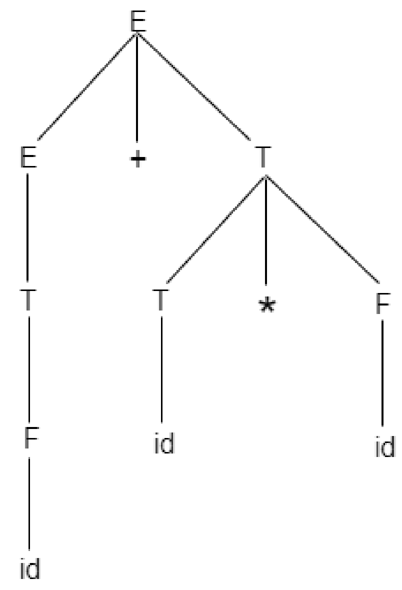
9.1.4 Abstract syntax tree design

In order to only maintain useful information and create logical error handling, an abstract syntax tree will be designed and implemented.

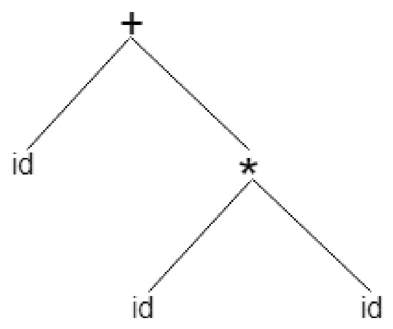
When designing a data structure for a translator it is important to recognize which features the structure should highlight or contain. This is why it is important to know the structural differences between a parse tree and an abstract syntax tree.

The most general difference is that in a parse tree, some nodes might be redundant, while abstract syntax trees eliminate all unnecessary information, and is thus a condensed but still parsable version of the parse tree. In abstract syntax trees, operations will be pulled up one layer and their operands will be their children nodes. [25]

Based on the pros and cons of each structure, an abstract syntax tree structure is chosen in order to not contain more information than needed.



(a) Parse Tree Example



(b) AST Example

Figure 9.1: Comparison between parse tree and AST [25]

The goal for the usage of the tree structure was to maintain useful information and create logical error handling. To do this the structure must live up to the following requirements:

- Variable types must be preserved, as well as the location of each declaration in the source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

It is now possible to look into the contextual constraints of the compiler.

9.2 Contextual Analysis

The purpose of the contextual analysis is to verify that the correctly written code also conforms to the contextual requirements of the language, such as declaration order, and type- and scoping rules. This is done by firstly traversing the AST, to build a symbol table, and later using the AST and symbol table to perform all the contextual requirement checks.

9.2.1 Scope

Most languages use the concept of scoping. A scope defines a set of variables that are all defined at the same level. Having multiple scoping levels allow the programmer to re-declare a variable in a new scope, while still being able to access upper-level scope variables.
The language developed in the project will have a simple scope system, which is a by-product of the simple block structure. The full set of scoping rules is defined in 8.3.1. There will be an upper scoping level for each block definition, and then one for the channel declarations, one for the blueprint definition and finally one for each defined procedure. This results in a scope tree with a maximum height of three.

9.2.2 Type Checking

Type checking is often integrated into the contextual analysis. This is the process of enforcing the restriction of types. A static type check is done at compile time, and a dynamic type check is done during run-time. Certain type checks require information only available at run-time, which is why dynamic type checking can be useful.

In this project, type checking will be fully done during compile time, although further development might require run time type checking. The language is a split between static and dynamically typed since the programmer is required to define supertypes while declaring new variables, which can be accomplished by grouping similar subtypes as children of their respective supertypes.

9.2.3 Chain checking

The system of establishing connections in the language requires that each connection is made between individual channels, or a group of individual channels to an element with that many inputs. The chain checking phase of the compiler should ensure that the connections made follow these bounds.

9.2.4 Dataflow checking

The concept of the language is for the user to define blocks with specific connection points, and connecting them into a graph that transforms a specified form of input into a corresponding output.
To ensure that every element will be able to fulfill its purpose, the flow checking phase of the compiler will take stock of every connection point that is declared, and check that they all are connected. Since the language

does not allow elements without an output, this also ensures that there is a connection from the input to output of any given block.
The flow checker should also enforce that an input can only be connected to one channel, while an output can be spread as widely as desired.

9.3 Code Generation

The important goals to achieve in code generation is, that the target code (code generated by the compiler), has the exact same meaning as the code compiled. It also has to be efficient in terms of CPU usage and memory allocation [26]. To avoid having to manage these details, Java was chosen as an intermediate compilation language which allows a more narrow focus on the code selection. To add functionality to the generated code, a Java-library will be created which will serve as the back-end for the language. This will be described further in the implementation part.

9.3.1 Running compiled Java code

After compilation from source code to Java, the programmer must also be able to execute the code. To execute the compiled code, the programmer will be able to use the intermediate Java library as an access point to the generated code.

To execute the code, the programmer will have two methods available for running the generated code. The first method is the $.train(Matrix\ inputs, Matrixtargets, intiterations)$ method, which allows the programmer to specify input and target data for the network to learn. The second method is the $.evaluateInputs(Matrixinputs)$ method, which allows the programmer to evaluate inputs on a pre-trained network, to get predictions based on the input data.
These methods are executed from a given Block defined within the source code. Specifically a "main block", which is a block where the blueprint has no parameters, and only one input and one output channel. It will be the responsibility of the programmer to run these methods on the appropriate block.

9.4 Symbol Table

The symbol table is an important link between compiler phases, that stores and shares relevant information. It might store information about scopes, variables within that scope and their types. [21]

The symbol table is typically implemented in one of four different data structures: List, Linked List, Hash Table, or Binary Search Tree. Each data structure has their own contextual advantages and disadvantages. hash tables, also called hash maps in Java, are the fastest option in the case of frequent insertion and look-up of entries. This will achieve the most optimal compilation efficiency [27]. The disadvantage of a Hash Table is the complexity of developing an optimal hash function. For its efficiency hash table is chosen as the data structure for the symbol tables of this compiler.

The compiler will generate multiple symbol tables corresponding to the syntax of a given code. One symbol table will be global and accessible to all scopes in the code. Besides the global table, a local symbol table will be created for each scope. As mentioned in Section 9.2.1, the language has a small amount of different scopes. All of the scopes will receive their own symbol tables, which is possible in a multi-layered symbol table structure.

Figure 9.2 shows how the symbol table's overall design looks, which gives an idea of how the symbol table design should be made. Primarily, a root table consisting of at least a single block. The blocks in the root table are references to tables for each individual block. All block tables consist of a single channel declaration scope, a single blueprint scope and an unspecified amount of procedures. In a way similar to blocks, these scopes each have their own representative tables. The channel scope has to have at least one output channel, but the other scopes can have as few or many variables as needed.

With knowledge about the structure and use of the symbol table it was possible to design a general model of how it should be structured. The traversal of the AST will now be described.

Symbol Table.png

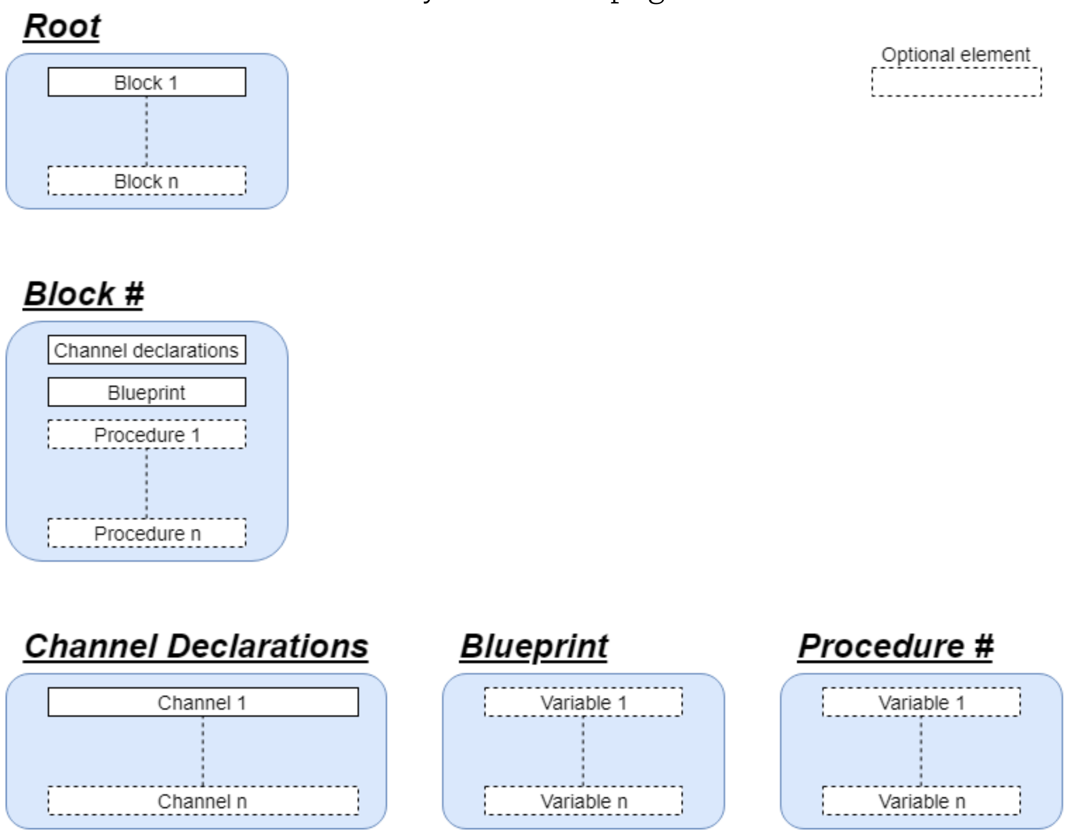


Figure 9.2: Symbol Table Example

9.5 Traversal

This chapter will focus on explaining visitor patterns and which solution has been chosen to be used in this project. The pros and cons will be discussed in order to state, why and how that choice was the better solution.

Visitor patterns can solve recurring design problems and they accomplish that by defining a separate visitor object, that implements an operation to be performed on elements of an object structure.

Instead of using a visitor pattern, it became clear, that the more traditional approach would be more suitable, due to the compactness of the resulting code, contrary to the visitor pattern which distributes the responsibility. The compact code is beneficial due to the small size of the proposed language and fairly limited number of nodes, which allows it to stay manageable.

The traditional traversal approach is essentially a big switch-statement on the type of the current node in the traversal of the AST, where functionality is added based on the type of node.

The traditional approach was heavily used before the invention of visitor patterns. If the language were to grow in size from further development, the traditional traverser would complicate the code since the switch-statement would grow increasingly larger and more complicated.

There is both pros and cons for both solutions which made it difficult to give a solution which will always be considered the best, but at this point in development, the traditional approach was chosen as the most clear and simple solution in this small language. This concludes the translator design, and the implementation of the compiler for the proposed language will be covered next.

Summary

This chapter gave an overview of how each compiler phase will be used. The tools JFlex and Cup were chosen to generate a scanner and parser respectively. Each of the phases is briefly described at a theoretical level, which forms the basis for the implementation. A proposed structure of a symbol table is explained as well as how the visitor pattern will be used.

With a general overview of each of the compiler phases and how the compiler of this project is going to be structured, it is possible to go in depth with the implementation of each of these phases.

Part IV
Implementation

10. Syntax Analysis Phase

10.1 Scanner

As mentioned in Section 9.1.3, the JFlex tool was chosen as the scanner generator for this project. This section seeks to describe the process of implementing the scanner for the proposed language, using JFlex. The scanner takes a source program as input, i.e. a stream of characters. Each *word* in the input is called a *lexeme*. The scanner's purpose is to convert the sequence of lexemes into a sequence of tokens, as described in Section 8.2.1.

10.1.1 JFlex

JFlex is used to generate a scanner for the language. To generate the scanner, the relevant information about the language is written in a specification file. The specification file consists of three parts, divided by %%: User code, options and declarations, and lexical rules. User code is the least significant part. This part will be copied directly to the top of the generated scanner class. This part consists of the package- and import-declarations.

Options and macros

The second section of the specification file holds the JFlex options as well as macro declarations. The set of options is code that will be included in the generated scanner class. Each line represents an option and is started with %. These include what the generated scanner class should be called, as well as member variables and functions that are used in scanner actions.

The specifications continue with macro declarations. Macros are shortened versions of regular expressions. Each macro consists of a macro identifier, followed by an equals sign (=), and then the regular expression, that the macro represents. Some examples of macros are as seen in Listing 10.1:

Listing 10.1: Macros example

```
Ident             =  [a-zA-Z_][a-zA-Z0-9_]*
NumLiteral        =  {IntLiteral}+(\.{IntLiteral}+)
IntLiteral        =  [0-9]+
New_line          =  \r | \n | \r\n | \n\r
white_space       =  {New_line} | [ \t\f]

Comment           =  {One_Line_Comment} |
                     {Multi_Line_Comment}
OneLineComment    =  "//"[^\r\n]* {New_line}
MultiLineComment  =  [/][*][^*]*[*]+([^*/][^*]*[*]+)*[/]
```

The macros consist of regular expressions for identifiers, numerals, newlines, white space, and comments.

Lexical rules

The lexical rules section, of the JFlex specification file, contains regular expressions and actions that are executed when the scanner matches the associated regular expression. The set of regular expressions is separated into five *groups*:

- *Keywords*, such as blocks, blueprints, and procedures.
- *Identifier names*
- *Literal numbers*, integers and doubles
- *Separators*, such as comma, semicolon, and parentheses.
- *Ignores*, such as white space and comments.

Summary

From the specification file, described above, the JFlex tool will generate a scanner class, that will take a stream of lexemes as input, and generate a stream of tokens as output. The tokens can be used by the parser to generate an abstract syntax tree, which will be described in the following section.

10.2 Parser

To generate the parser portion of the compiler, the parser generator CUP was chosen for this project. This section will cover how the CUP specification for the project is set up to generate a parser to take the token-stream supplied by the scanner, in addition to generating a valid abstract syntax tree for the language.

10.2.1 CUP

To generate a parser with CUP, all necessary information about the language is written in a specification file. This file covers the names of all terminals and non-terminals, as well as LALR(1) compliant production rules in basic BNF. Each production rule is outfitted with a block of associated action code. There is also the option for preemptively specifying or overwriting code for the parser that is to be generated.

Parser code

The parser code section of the CUP specification covers code to be included in the parser that is not attached to any production rules. In this project's implementation, it includes the root node for the AST that the parser will be building. Furthermore, it also includes functions to create or access this node. Error handling for testing the parser is also included under parser code.

Production rules

The production rules in the CUP specification file is located after the keyword *starts with 'nonterminal'*, where the non-terminal in this case is replaced with *prog*. Following this, the production rules show the correspondence between non-terminals and each of their production rules.

Listing 10.2: Production rules example

```
statement   ::= declaration
                 | ID optidstatement
                 | LPAR optlparstatement
                 | THIS DOT ID optthisstatement
                 | builddeclaration chain
                 |
                 ;
```

Listing 10.2 shows, each possible pattern that the non-terminal can take depending on its follow set in order to validate the code written in the language. It is constructed in such a way, that the non-terminals are displayed in lowercase and the terminals are displayed in uppercase. The original non-terminal is followed by *::=*, which indicates the beginning of the non-terminal's rules. Then each extra rule is initiated with the or-symbol | followed by a set of terminals and non-terminals. When the rule-set is complete, it ends with a semicolon ;.

Action code

Action code is code, which is called immediately after performing a reduction with the associated production code [28]. The action code is used to create a data structure that matches how the parser validates the written code. This can then be used to create a parse tree and an abstract syntax tree. As stated in Section 9.1.3, no parse tree will be generated, but it is however used to create an AST.

As CUP is an LALR parser, it parses bottom up. This means that the resulting tree is built from the leaves up, where every produced node passes themselves on to their parent-to-be. This means the action code of any given branch of the tree can work with any children they may have, but not their parent.

Listing 10.3: Action code - Example 1

```
channelelement ::= THIS DOT ID:id
             {: RESULT = new
                 SelectorNode("this").adoptChildren(new
                 SelectorNode(id)); :}
             | ID:id dotid:dotid
             {: RESULT = new
                 SelectorNode(id).adoptChildren(dotid); :}
```

The action code in Listing 10.3 is built into the production which enables the specified code to be executed. In this listing, a node representing the essence of the production is instantiated, and the other relevant information is attached to it. This is then all passed to the *RESULT* in order to connect the branch further up the tree.

Listing 10.4: Action code - Example 2

```
| element:elem parclosestatement:chainOrGroup
{:
   // Group connection
   if (chainOrGroup instanceof StorageNode) {
      AbstractNode group = chainOrGroup.getNodes()[0];
      AbstractNode chain = chainOrGroup.getNodes()[1];

      group.adoptAsFirstChild(elem);

      RESULT = chain;
   } else {
      RESULT = chainOrGroup.adoptAsFirstChild(elem);
   }
 :}
```

Listing 10.4 shows a production with a bit more action code going on. The depicted production is one of the options for the non-terminal *optl-parstatement* (Optional parentheses statement). It handles statements that start with exactly one left parenthesis, which is a case that doesn't immediately show if this is a chain or group connection being parsed.

The non-terminal *parclosestatement* can either return a ChainNode for a chain connection, or for group connections a *StorageNode* containing the *GroupNode* and its associated *ChainNode*. The element parsed in this production is the first element of either case. For the chain connection case, this is accomplished simply by adopting it as the first child, but with the group connection case, it is added to the group, which as a whole already makes up the 'first element' of the chain. In either case, the now completed chain is the node that is actually returned up through the tree.

Abstract syntax tree

The AST has been implemented using a node structure where some node types extend one another. This structure will define the primary implementation of the AST.

The nodes of the AST have varying quantities of information they need to contain. Because of this, inheritance is an ideal way of categorizing them. The node inheritance structure can be seen in figure 10.1.

The *Node* interface contains all the essential base methods for a node object implementation expected by the compiler and *AbstractNode* contains

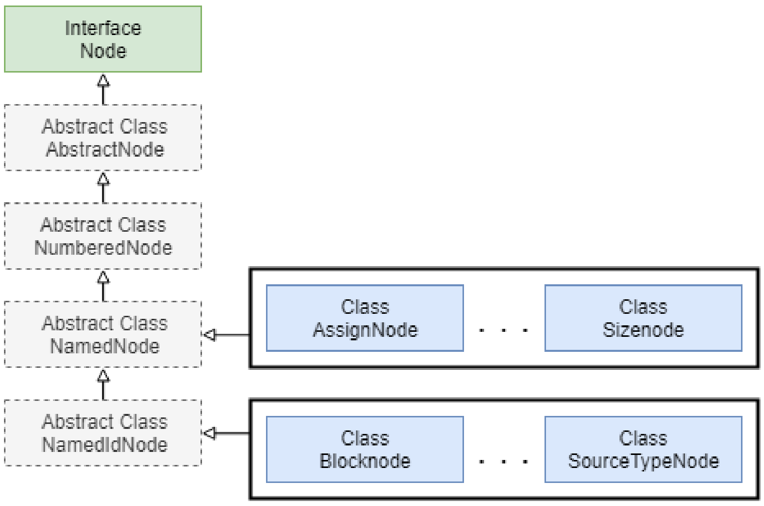


Figure 10.1: The inheritance of the various nodes in the compiler.

the total set of those methods fully implemented. The remaining abstract classes contain expanded functionality. *NumberedNode* also has the functionality to contain an integer, *NamedNode* can contain a *String* name, and the node with the most functionality is *NamedIdNode* which also contains an *id* stored as a *String*.

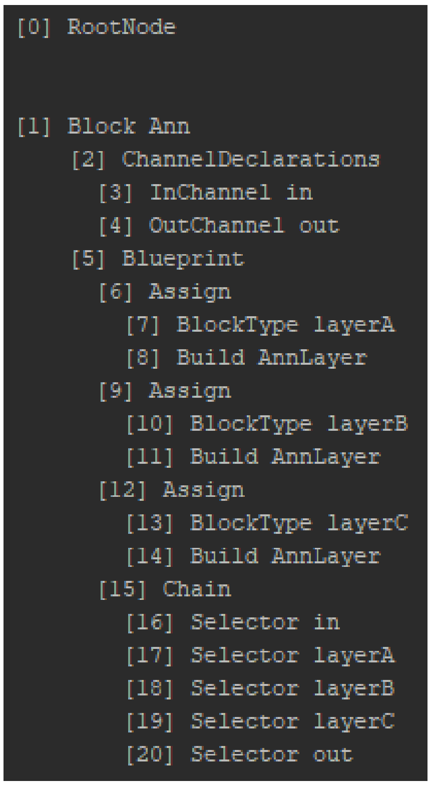
In the implementation of the compiler, the only classes being extended directly by a non-abstract class are both *NamedNode* and *NamedIdNode*.

With a fully developed node structure, it is possible to apply the node structure in the action code of the parser, in order to create the AST. For development purposes and testing, the tree is displayed in the application console when it is generated.

Listing 10.5: Nodes included in action code

```
channel                 ::= CHANNELIN
                        {: RESULT = new InChannelNode("NoIden"); :}
                        | CHANNELOUT
                        {: RESULT = new OutChannelNode("NoIden"); :}
```

Listing 10.5 shows how the node structure can be used within the action code of the CUP specification. With node structure implemented in all non-terminals with relevant information, it is possible to create an AST as seen in Figure 10.2.
But to make all this work, some way of generating the scanner and parser had to be implemented, and that will be introduced using ANT builds.



```
[0] RootNode


[1] Block Ann
    [2] ChannelDeclarations
      [3] InChannel in
      [4] OutChannel out
    [5] Blueprint
      [6] Assign
        [7] BlockType layerA
        [8] Build AnnLayer
      [9] Assign
        [10] BlockType layerB
        [11] Build AnnLayer
      [12] Assign
        [13] BlockType layerC
        [14] Build AnnLayer
      [15] Chain
        [16] Selector in
        [17] Selector layerA
        [18] Selector layerB
        [19] Selector layerC
        [20] Selector out
```

Figure 10.2: Abstract syntax tree console example

10.2.2 Ant

Before they can be used in the compiler, the scanner and parser need to be generated according to their specifications in JFlex and CUP respectively. For this purpose, Apache Ant is used in the project. Ant can be used to build and compile Java code from various sorts of sources. In this case, it is used to build the code for the scanner and parser according to their JFlex/CUP specifications.

Summary

This chapter went through the implementation of both the scanner and parser. This involved how JFlex and cup were configured to scan and parse a program, and then use action code when parsing to generate an AST. Recognition patterns in the form of regular expression were given to JFlex to generate a scanner, and the unambiguous production rules were given to Cup to generate a parser.

With the scanning and parsing phases of the compiler in place, the next step is the contextual analysis on the resulting abstract syntax tree.

11. Contextual Analysis Phase

11.1 Symbol Table

The class *SymbolTable* is essential to this project's developed compiler. In the project, it's implementation utilizes several other unique classes within the project. The symbol table only has the necessary methods, which ensures uniform usage of objects of the class.

11.1.1 Symboltable.java

The symbol table is implemented as an interface, as seen in Listing 11.1. This is because there are many different ways to implement a symbol table data structure.

Listing 11.1: Symbol table interface

```java
public interface SymbolTableInterface {
    void openBlockScope(NamedNode node);
    void openSubScope(NamedNode node);
    BlockScope getBlockScope(String id);
    BlockScope getLatestBlockScope();
    void insertVariable(NamedNode node);
    void reassignVariable(NamedNode assignNode);
    boolean isPredefinedOperation(String operation);
}
```

In the compiler, there exist a single SymbolTable data structure, which utilizes an object of the class *NamedTable* to store and retrieve block scopes as seen in Listing 11.2. These block scopes contain subscopes within another *NamedTable*. The subscopes contain the symbols of their scope. This implementation will be described in Subsection 11.1.4. A NamedTable is an extension of *HashMap* where the key is always a string object.

Listing 11.2: SymbolTable class implementation

```java
public class SymbolTable implements SymbolTableInterface {
    private NamedTable<BlockScope> blockTable = new
        NamedTable<>();
    ⋮
```

11.1.2 Enum Switches

The implementation of nodes in the compiler heavily relies on the extension of other classes. As a result, a simple way of specifying a type of node is by using the *instanceof* operator in Java. The language does not allow a switch statement to utilize any similar functionality to the *instanceof* operator. This is an issue, as a large amount of implemented nodes can smartly be handled by a switch statement.

To deal with this limited functionality, every class inheriting from NumberedNode is expected to contain a *NodeEnum*. These enums are a one-to-one way of specifying a node's type. This allows easy utilization of the switch statement for nodes, as can be seen in Listing 11.3.

Listing 11.3: Switch structure

```java
switch (node.getNodeEnum()) {
    case ROOT:
    case GROUP:
    ⋮
    case OPERATION_TYPE:
    break;

    case BLOCK:
        this.currentBlockScope =
            this.symbolTableInterface.getBlockScope(id);
    break;

    case PROCEDURE:
    this.currentSubScope =
        this.currentBlockScope.getProcedureScope(id);
    break;
    ⋮
    default:
        throw new RuntimeException("Unexpected Node");
}
```

11.1.3 Nametable

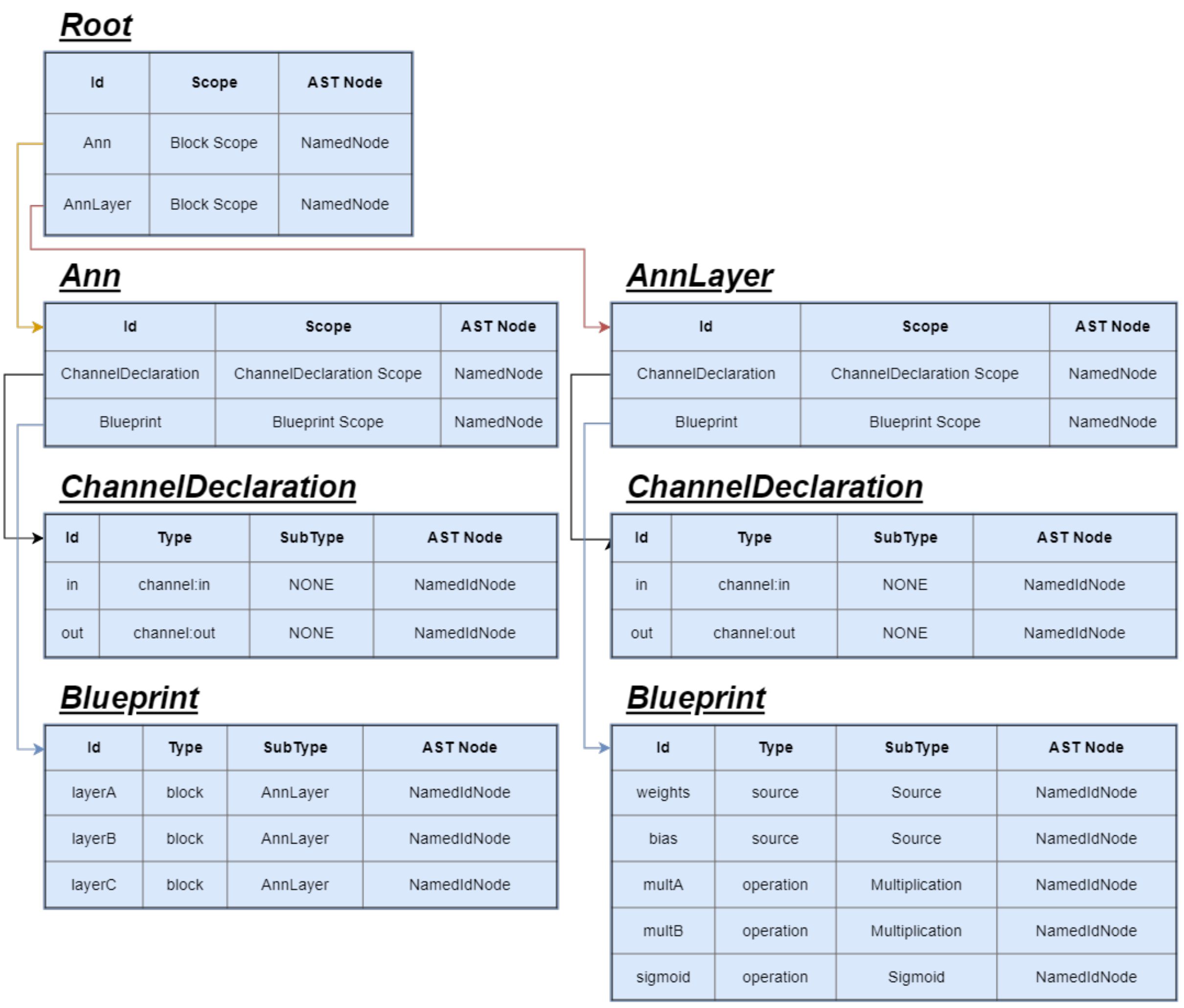
To implement the symbol table, a new *Nametable<T>* class was implemented as seen in Listing 11.4. This class functions very similarly to a normal Java map, however, it does have some notable differences. Most noticeably, it has limited functionality relative to a traditional Java map, and because of this, it is not interchangeable with any other map due to not implementing the *Map<K,V>* interface.
The nametable contains a *MapLatest<K,V>* reference which is an interface. This interface extends the *Map<K,V>* interface, and adds a new *getLatest()* method which retrieves the most recently added element of the map.
The *HashMapLatest<K,V>* class implements this interface.
The lack of access to generics and all method when using *NamedTable* is a limitation to specify the functionality and responsibility of the class.

Listing 11.4: Implementation of NamedTable class

```java
public class NamedTable<T> {
   private MapLatest<String, T> table = new HashMapLatest<>();

   public T getEntry(String name) { return table.get(name); }

   public T getLatest() { return this.table.getLatest(); }

   public T setEntry(String name, T entry) { return
      this.table.put(name, entry); }
   ⋮
```

11.1.4 Diagram

Figure 11.1 shows, how the final version of the symbol table has been implemented. It is a more detailed version of Figure 9.2 in the sense, that now there has been added details about the specific identities, scopes, types, subtypes, and the node structure. Furthermore, it is now visible that all tables are intertwined. An example of how the symbol table is displayed in the console can be seen in Appendix D.



Root

Id	Scope	AST Node
Ann	Block Scope	NamedNode
AnnLayer	Block Scope	NamedNode

Ann

Id	Scope	AST Node
ChannelDeclaration	ChannelDeclaration Scope	NamedNode
Blueprint	Blueprint Scope	NamedNode

ChannelDeclaration

Id	Type	SubType	AST Node
in	channel:in	NONE	NamedIdNode
out	channel:out	NONE	NamedIdNode

Blueprint

Id	Type	SubType	AST Node
layerA	block	AnnLayer	NamedIdNode
layerB	block	AnnLayer	NamedIdNode
layerC	block	AnnLayer	NamedIdNode

AnnLayer

Id	Scope	AST Node
ChannelDeclaration	ChannelDeclaration Scope	NamedNode
Blueprint	Blueprint Scope	NamedNode

ChannelDeclaration

Id	Type	SubType	AST Node
in	channel:in	NONE	NamedIdNode
out	channel:out	NONE	NamedIdNode

Blueprint

Id	Type	SubType	AST Node
weights	source	Source	NamedIdNode
bias	source	Source	NamedIdNode
multA	operation	Multiplication	NamedIdNode
multB	operation	Multiplication	NamedIdNode
sigmoid	operation	Sigmoid	NamedIdNode

Figure 11.1: Implementation of symbol table

11.2 Scope Checking

Scope checking is implemented by doing an iteration over the AST in pre-order, and each time a node is met that is dependent on some scoping, it updates the current scope or checks whether the scoping is correct. If on the contrary, the scoping is incorrect, it throws an exception. The reasoning behind doing it in pre-order instead of post-order is that pre-order follows the program sequentially and allows for entering a specific scope before checking whether the contents of that scope are correct.

A specific scope checking visitor is implemented to traverse the AST and act in relation to which type of node is met. This switch structure was described previously in Subsection 11.1.2. The actions it takes are partially described both formally and informally in Subsection 8.3.1.

The *ScopeCheckerVisitor* contains three fields. A *symbolTableInterface* which allows access to the symbol table, the current block scope and the current subscope which allows the ability to determine whether something has been declared within the current scope. These current scopes are updated as new scopes are entered, which appears from Enumeration 1 through 4 where if these nodes are met, a new scope is entered and thus, the 'current scopes' need to be updated. Enumerations 5 through 7 describe the nodes that need a scope check, and what concepts are to be checked.

1. BLOCK: When entering block, update current block-scope.
2. PROCEDURE: When entering procedure, update current subscope.
3. BLUEPRINT: When entering blueprint, update current subscope.
4. CHANNEL__DECLARATIONS: When entering channel declarations, update current subscope.
5. DRAW, BUILD: Ensure that the block trying to be built or drawn exists within the current root-scope, otherwise check it as a predefined operation or source. If this is not the case, a *NonexistentBlockException* is thrown.
6. PROCEDURE__CALL: Ensure that the procedure being called has been declared within the current block scope, otherwise throw a *ScopeBoundsViolationException*.
7. SELECTOR:
   - If 'this'-selector is specified, it is verified that the channel it selects is declared within the current block's channel declarations.
   - Procedure-call is handled in the type checker.
   - Ensure that a variable being selected exists within the current scope, and if it doesn't, check if it is a channel, which would have been declared in the channel declaration subscope.

11.3 Type Checking

The type checker is implemented as a tree walker which extends the abstract *ScopeTracker* tree walker. This allows for ease of tracking, of the current block scope and subscope which is used for the type checking phase.

The type checker utilizes the implemented class *TypeSystem* for most of its operations. A UML diagram of the class can be seen in Figure 11.2, which describes the fields and public methods of the class.
The most important methods for the type checking phase are the two methods: *getSuperTypeOfNode()* and *assertEqualSuperTypes()*.

Method *getSuperTypeOfNode()*: Takes a node of the AST as input, and returns the supertype of that node.

Method *assertEqualSuperTypes()*: Takes two nodes of the AST as input, and asserts that they have the same supertype, otherwise an exception is thrown.

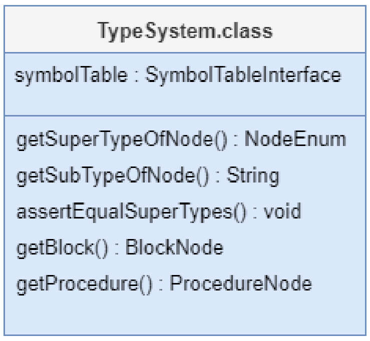

TypeSystem.class
symbolTable : SymbolTableInterface
getSuperTypeOfNode() : NodeEnum
getSubTypeOfNode() : String
assertEqualSuperTypes() : void
getBlock() : BlockNode
getProcedure() : ProcedureNode

Figure 11.2: UML diagram of the TypeSystem.class

The type checker walker only care about 3 different cases, consisting of 4 different nodes of the AST.

1. CHAIN: When entering a chain, it is checked that all the chain elements uphold the type rules defined in Section 8.3.3
2. ASSIGN: Whenever the programmer assigns a value to a variable, the supertype of both the variable and the value are compared and expected to be equal.
3. BUILD, PROCEDURE_CALL: When a *call* is encountered, which is either a build-statement or a procedure call, the amount of actual and formal parameters from the caller and callee is compared followed by a comparison of the supertypes.

With these cases checked, subsequent phases know that all variables, parameters, and chain connections are of correct supertype for their usage within the program.

11.4 Semantic Checking

The semantic analysis primarily goes in depth with the concepts of a programming language which isn't easily described or handled by any earlier stages of a compiler. In this case, most of these cases are in regards to subtyping and flow-checking.

With that goal in mind, it's needed to traverse through a tree structure using a new semantic analysis visitor.

11.4.1 Chain Connection Size

In the type checking phase, it was verified that all elements of a chain *could* be placed within the chain and in the given position. But it was never verified that the connection made *sense*.

This phase will also look at the input and output sizes of the elements within the connections. There are only two different cases to look at:

1. Group connection: A group connection is a short-hand notation for connecting multiple elements into a single element. With groups, it's important to verify that all elements of the group only have one output channel, since otherwise it's undefined which one to connect, and lastly, the element connected to should have exactly as many input channels as there are elements in the group for the same reason. The group will always be the first element in the connection.
2. Short-hand chains: A short-hand chain in this context is of the form $x \rightarrow y \rightarrow z$. With these chains, the verification that all elements between the first and last, only has one input and output channel is made, to avoid connection ambiguity. The first element should only have one output but can have multiple inputs, since the inputs are not part of the chain statement - The same principle reversed applies to the last element and its input channels.

11.4.2 Build Recursion

When block X builds block Y, and block Y, in turn, builds block X within its blueprint, a block recursion happens - and the program is undefined since it would never finish the compilation.

The proposed language is finite with no branching involved, which means that building a block with the same parameters will always result in the same product of such action - This information is useful for detecting build recursions, since if the same build call happens as a sub-call of itself, there will without a doubt be recursion involved.

In order to detect these recursions, the compiler keeps track of the call stack, and if it detects that a new sub-call is identical to a previous call,within the call stack, a recursion has happened, and the compiler will display an appropriate error message.

Since the call stack can potentially grow very large, it's not a feasible idea to compare every new call with every parent call to detect if they are identical. Therefore the compilers call stack is a data-structure which utilizes both a set and a stack. The stack is used to know the order of insertion and extraction, while the set is used to perform $O(1)$ comparison to all previous calls, which allows the compiler to keep the stack structure, and have constant time comparisons when creating a new sub-call.

11.4.3 Dataflow checking

The flow checking phase of the compiler ensures that all channels of each build element are connected, to ensure a flow from the input to the output of the generated neural network. This is achieved by collecting information about all build statements and all connections, and comparing them to each other.

The flow checker walks the AST recursively, following the call structure of the code, and bringing parameter variables along by value. For every build statement encountered, all of its mychannels are noted down, together with the toString() representation of the BuildNode in question as an id of the specific instance of the block/operation. These notes are added to a Set according to whether it is a in- or out-channel. The reason a Set specifically is used will be explained later.

Whenever a connection chain is encountered, each variable involved is tracked back to the BuildNode the involved element originates from. Each connection established is then noted down in the same format as the build statements.

At the end of the walk of any blueprint scope, before returning back to where the recursive walk called it from, the connections of this instance are evaluated. The collections of channels declared and connections established are compared, removing channels from the Sets of existent channels as they get connected. Because a Set is used, if any element is attempted to be removed twice, a boolean 'false' is returned. This is used to throw an error if any in-channel was connected more than once.

When all the connections are evaluated, any channels remaining in the Sets will not have been connected to anything. This is a valid reason for throwing an error.

Summary

This chapter described how the different compiler elements inside the Contextual Analysis Phase were implemented. A symbol table structure was made to allow keeping track of variables and scoping. It was generated by traversing the AST and switching on the type of the node. After an initial symbol table has been filled the AST is traversed both to make scope checking and type checking but now referring to the AST. Lastly, the semantics of the language are checked.

With the checking of the contextual analysis phase complete, it's possible to address code generation with the intention of compiling from the project language to an intermediate Java library.

12. Code Generation Phase

This chapter seeks to describe the process of implementing the final phase of the compiler, the code generator. The chapter will go through the decisions made throughout this phase and will highlight some of the main features in the implementation.

As described in Section 9.3, the purpose of the code generator is to generate a program in a target language, which is equivalent to the source program. In the first two phases of the compiler, an AST is generated and decorated. Code generation follows a depth-first traversal of said AST. [21]

12.1 Choice of intermediate compilation language

When deciding on an intermediate compilation language, several issues have to be considered. One main issue is how to model high-level structures in a low-level language. Three sub-issues are considered:

- **Code selection:** The process of determining in what order the target machine instructions will be used to implement each phrase in the proposed language. [21]
- **Storage allocation:** Storage allocation describes how each variable in a program is stored. This means choosing between static allocation and stack allocation, as well as choosing how to handle garbage collection. [21]
- **Register allocation:** Register allocation is only relevant for register-based machines. The issue with register allocation is figuring out how to use registers efficiently to store intermediate results. [21]

An example of a language that trivializes these processes, is Java. Java compiles to the Java Virtual Machine (JVM), which handles both storage and register allocation on its own. For this reason, the proposed language compiles to Java, which then compiles to the JVM. Therefore only code selection will have to be worried about. The next section will discuss how the proposed language will be compiled to Java.

12.2 Intermediate Java library

To ease the process of generating code from the proposed language to Java, an intermediate library is written in Java. The idea is to create a library in Java with the same constructs and structure as the proposed language, such that the compiler can generate code to the library directly.

The library consists of several Java classes. The library contains classes for blocks and channels as well as classes for basic operations to be used in the proposed language. These will be described later. The implementation of the library is viewed as a part of the implementation of code generation as the code will be generated to said library.

To highlight some of the similarities between the proposed language and the intermediate Java library, the block 'A_plus_B_mult_B' is used as an example. The graph to visualize the block is shown in Figure 12.1.

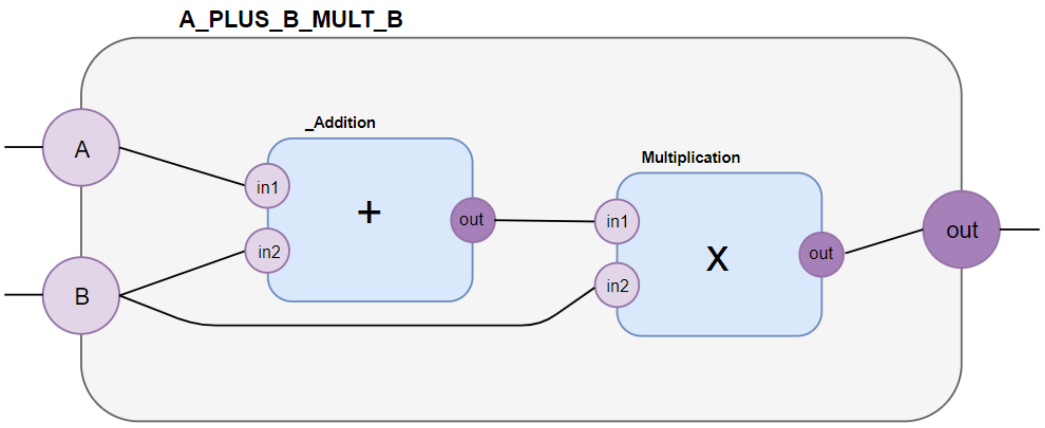


Figure 12.1: Illustration of structure of A_PLUS_B_MULT_B block

The block takes two inputs, gives a single output, and consist of two operations. As the name describes, the block takes two matrices A and B as inputs, adds them together, and finally multiplies the resulting matrix with the input matrix B.

The operation shown in the example, as well as any other operations featured in the proposed language, are all implemented in the intermediate Java library. These are divided into unary- and binary operations. Examples of these include addition as a binary operation and matrix transpose as a unary operation. Furthermore, each type of operation is divided into matrix-wise and unit-wise operations, with matrix-multiplication being a matrix-wise operation, and matrix subtraction being a unit-wise operation.

Listing 12.1 shows how the block is implemented in the proposed language, and Listing 12.2 shows how the block would be implemented in Java, using the intermediate library. The two main differences in the appearance of the code are seen in channel declarations and in connections. In lines 2-4 in Listing 12.1, and in lines 4-6 in Listing 12.2 it's seen that the proposed language allows for a cleaner and more intuitive way of declaring channels, as it is similar to declaring a variable in for example Java. In lines 10-11 in Listing 12.1, and in lines 12-13 in Listing 12.2, shows the difference in how connections are made in the two languages. It's apparent that the proposed language uses a more readable solution, as the arrow notation is used to connect blocks.

Listing 12.1: Implementation of A_plus_B_mult_B block in proposed language

```
1   block A_plus_B_mult_B {
2      mychannel:in A;
3      mychannel:in B;
4      mychannel:out out;
5
6      blueprint() {
7         operation add = build _Addition();
8         operation mult = build Multiplication();
9
10        this.A -> add.in1;
11        this.B -> add.in2;
12
13        add.out -> mult.in1;
14        this.B -> mult.in2;
15
16        mult.out -> this.out;
17     }
18  }
```

Listing 12.2: Implementation of A_plus_B_mult_B block in Java library

```java
public class A_plus_B_mult_B extends AbstractBlock {
   public A_plus_B_mult_B() {
      // Channel declarations
      this.addNewInputLabel("A", new ListChannel());
      this.addNewInputLabel("B", new ListChannel());
      this.addNewOutputLabel("out", new ListChannel());

      // Blueprint
      Operation add = new _Addition();
      Operation mult = new Multiplication();

      this.connectTo(add, "A", "in1");
      this.connectTo(add, "B", "in2");

      add.connectTo(mult, "out", "in1");

      this.connectTo(mult, "B", "in2");

      mult.connectTo(this, "out", "out");
   }
}
```

12.3 Code Selection

The language utilizes the intermediate library for code generation, and therefore the product of the compiler is a set of Java classes, and how these are generated will be explained in the following sections.

12.3.1 The Java class

An ordinary Java class structure can be seen in Figure 12.2, this illustration shows the 5 common sections of a Java class. These sections are relevant to the code generation phase since these are what must be generated as a result of the input code.

- **Class package:** This section tells the class which package it is contained within.
- **Imports:** This section tells the class where to find all of its dependencies.

- **Fields:** This section declares all fields within the class, which can be viewed as information relevant to the class. The fields can be accessed globally within the class itself.
- **Constructor:** This section declares the constructor of the class. The purpose of the constructor is to define how the class should be initialized.
- **Methods:** This section declares all methods of the class, they can generally be public or private to the class.

While all of these sections carry information for the class, they might not all be relevant in every case. Specifically the section of "Local Fields" will not be relevant while translating from source code to Java. The reason behind this is that the language does not support variables which can be accessed globally within the block, except for channels and these are handled by the intermediate library.

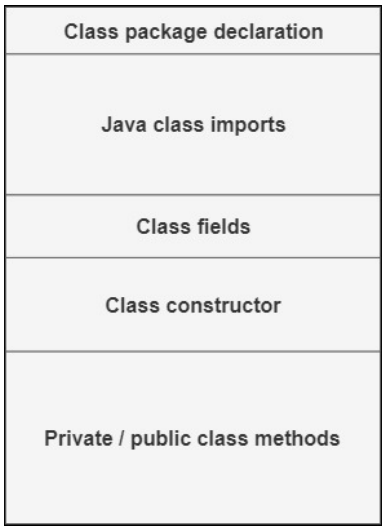


Figure 12.2: Illustration of an ordinary Java class structure

In Java, each public class has to be in its own file, with the same name as the class. Because of this structure, it is required to generate a separate file and class, for each block the programmer defines within their code.

12.3.2 Representing a Java class

To build an internal representation of a class, the class "BlockClass.java" has been created, which is a data structure used within the code generation phase to build a single class. A diagram of the class fields and methods can be seen in Figure 12.3

A BlockClass object has several instances of the class CodeScope. A Code-Scope represents a single subscope within a block, such as a blueprint and procedures, and will be translated into a Java class method of the same name within the BlockClass. A diagram of the CodeScope class can also be seen in Figure 12.3.

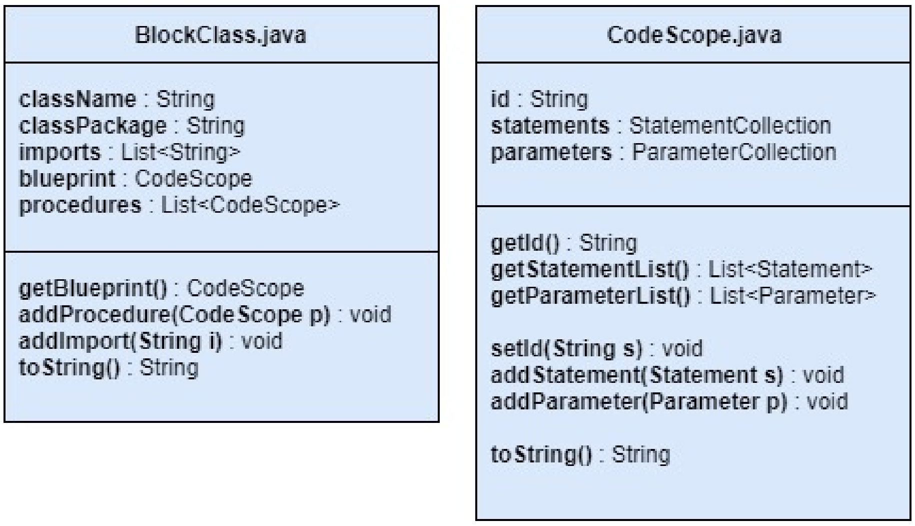

BlockClass.java
className : String
classPackage : String
imports : List<String>
blueprint : CodeScope
procedures : List<CodeScope>
getBlueprint() : CodeScope
addProcedure(CodeScope p) : void
addImport(String i) : void
toString() : String
CodeScope.java
id : String
statements : StatementCollection
parameters : ParameterCollection
getId() : String
getStatementList() : List<Statement>
getParameterList() : List<Parameter>
setId(String s) : void
addStatement(Statement s) : void
addParameter(Parameter p) : void
toString() : String

Figure 12.3: BlockClass.java and CodeScope.Java class diagrams

When traversing the abstract syntax tree, a BlockClass object is instantiated and stored in a list. Such an object is initiated only once for each block declaration made within the source program. While traversing the abstract syntax tree, statement nodes are added to an internal list of the current BlockClass, which each represents a small part of the full block and its final functionality. A full list of statement object can be seen in Appendix G.

An example of a statement object is the "InitBuild.java" statement. The purpose of this statement is to represent a single build statement, such as "$build\ Multiplication()$".
To create this statement, the compiler makes the call
"$blockClass.addStatement(new\ buildInit(buildId,\ buildParameters))$", where the buildId is the ID of whatever is being built, and the buildParameters is a list of all parameters used in the build statement.
The example would compile into: "$(Operation)\ new\ Multiplication()$"

12.3.3 Generating the Java class files

After the full abstract syntax tree has been traversed, a complete list of BlockClass objects has been created with all the required information contained within them. The list of BlockClass objects are then iterated by the compiler, and by writing the .toString() result to a file with the same name as the block, a Java class of the defined block will be created.

The generated Java block classes can then be accessed like any other Java class, in any Java project, then be trained, by using the "block.train(...)" method, and used to evaluate inputs and produce predictions based on data, by using the "block.evaluateInputs(...)" method, to show the performance.

Autogeneration example

An example of the code generated by the compiler from a simple Layer block can be seen in Listings 12.3 and 12.4.

Some of the most prominent effects of the translation, is that all subscopes - blueprint and procedures - are represented as methods, which are called when required. When a build is used directly within a chain, a temporary and unique variable must be created, before the element can be used multiple times within connection statements.

Listing 12.3: A block with the name Layer implemented in the language

```
block Layer {
   mychannel:in in;
   mychannel:out out;

   blueprint() {
      source weights = build Source([1000, 1000]);
      source bias = build Source([1000, 1000]);

      operation mult = build Multiplication();

      (this.in, weights) -> mult;
      (mult, bias) -> build _Addition() -> build _Sigmoid() ->
         this.out;
   }
}
```

Listing 12.4: Auto generated code of the block Layer

```java
public class Layer extends AbstractBlock {

   public Layer() {
      this.blueprint();
   }

   private void blueprint() {
      try {
         this.addNewInputLabel("in", new ListChannel());
         this.addNewOutputLabel("out", new ListChannel());
         Source weights;
         weights = (Source) new Source(new Pair<Integer,
            Integer>(1000, 1000));
         Source bias;
         bias = (Source) new Source(new Pair<Integer,
            Integer>(1000, 1000));
         Operation mult;
         mult = (Operation) new Multiplication();
         mult.receiveGroupConnection(this.getChannel("in"),
            weights);
         Block __Addition_514;
         __Addition_514 = new _Addition();
         Block __Sigmoid_311;
         __Sigmoid_311 = new _Sigmoid();
         __Addition_514.receiveGroupConnection(mult, bias);
         __Sigmoid_311.receiveGroupConnection(__Addition_514);
         __Sigmoid_311.connectTo(this.getChannel("out"));

      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }
}
```

Summary

This chapter went through the choice of the target language for the code generation where Java was chosen. Furthermore, the Java-library that should serve the functionality to the language was described, as well as the parallels between the proposed language and the Java code that is generated from it.
Finally, the code selection process was described, where the generated Java classes were defined as a result of a toString() method, on its inter-

nally created class-representation by the code generation visitor.

Having gone through the implementation of the different phases, Figure 12.4 illustrates an overview of the compiler and all its phases. The following chapter describes how the quality of the software implementation is ensured.

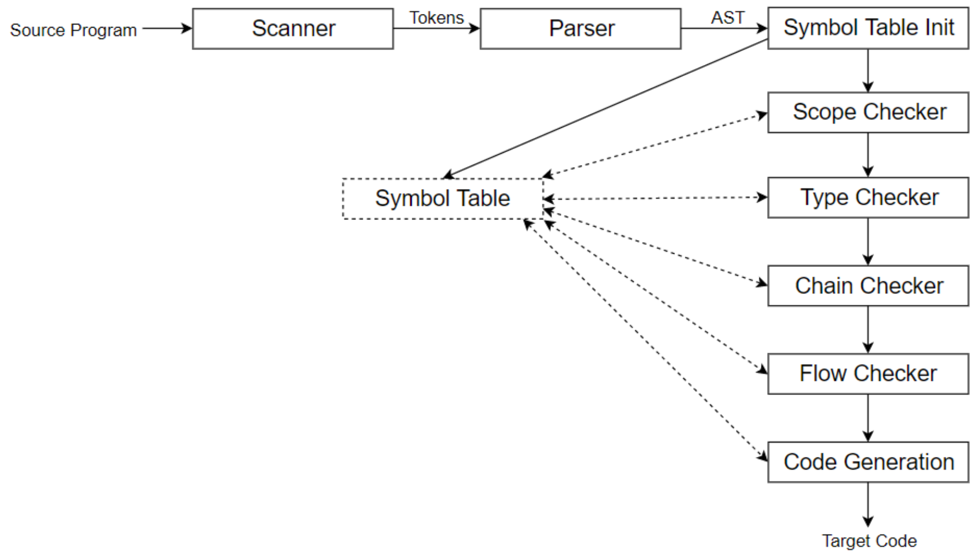


Figure 12.4: Overview of compiler phases for the proposed language

13. Software Quality

To improve the overall software quality, a set of methods can be used. Testing and documentation are central ways of doing this. A test is an experiment to determine what the outcome of a program's execution is. The test will pass if the outcome matches the expectations, and the converse is true if the test fails [29]. When implementing a large program, software documentation is an important part of improving the quality of the product by increasing ease of maintenance and further development [30].

13.1 Unit Tests

Unit tests test small and isolated components in the program. A result of its size and isolation is that when tests fail and uncover some unintended behavior, it is easier to pinpoint the exact piece of code that causes this behavior [29].

Generally, testing can be either manual or automated. Manual testing involves a programmer running the program and evaluating the outcome. This process can be time-consuming, error-prone, and repetitive. Automated testing, contrarily involves the computer running parts of the program and comparing the result to a predefined expected result. When the tests have been specified once, they can be run repeatedly, which adds a certain level of continuous correctness, even after refactoring parts of the code. [29]

The tool chosen to implement automated testing is the testing framework JUnit, which is an annotation-based tool. An example of a unit test written in JUnit appears in Listing 13.1. The '@Test' annotates that the following method is a test. A block scope is created with identifier 'b'. The 'getId' method in BlockScope is tested by asserting that it returns the expected id. An assertion is JUnit's way of making a contract that expects some specific outcome, which in this case is that the return value of getId is equal to 'b'.

Listing 13.1: Example of unit test written in JUnit

```java
@Test
void getIdTest() {
   BlockScope b = new BlockScope("b", node);
   assertEquals(b.getId(),"b");
}
```

Unit tests, such as the one previously described, are classified as white box tests. This entails that the programmer has knowledge about the unit being tested, and from that knowledge, can extract information about edge-cases, the extent of tests to be written and the coverage of tests. Test coverage is an important trait because it is a measurable way of determining the usefulness of a test suite. Though even with the measurability of test coverage, it won't ensure the absence of bugs, only reveal their existence. [29]

13.2 Program Tests

In contrast to unit tests, program tests test larger components in co-operation with each other. To ensure correctness and allow change and development of the compiler a test suite of program tests was developed. The tests run test-programs through the compiler and then expect some outcome, which determines the result of the test.

The compiler reads a program and moves it through its phases, which each ensures that some specific part of the compiler is correct. It is necessary to both test that correct programs are accepted by the compiler, and that incorrect programs return errors from the expected phase in the compiler. This is done by splitting the test suite into a number of expected true tests, and a number of expected false tests.

By continuously adding tests throughout the development of the compiler, the process becomes partially test-driven. When adding a test for something which is expected to pass or fail but doesn't, it drives the development to add that functionality in the compiler.

The test-programs are structured into individual text files and further into folders based on whether they are expected to be true or false, and which part of the compiler they test. To test each of these individual test-programs, a dynamic test is generated for each of them, which tries to compile them and asserts that the result of the compilation is what was expected. The mass-generation of the dynamic tests is made in test factories. A test factory must return a stream, collection or the likes of DynamicTests. When running the tests, these tests will be generated dynamically by the test factories. [31]

Listing 13.2 shows the implementation of a test factory for generating tests that compile programs that are expected to be correct. On line 4 it references the folder with all the 'ExpectTrue' program-files. From this it creates a list of each of the programs on line 6. On line 8 it returns the stream of dynamic tests, which is created by using a lambda. The static dynamicTest method needs a name and then an executable as parameters. The executable is the content of the dynamic test, which is expressed as a lambda which asserts that compiling the given program will be successful.

Listing 13.2: Use of TestFactory to test expected correct tests

```
 1 // Test all positive files
 2 @TestFactory
 3 Stream<DynamicTest> positiveFiles() {
 4    File trueFolder = new File("tests/ExpectTrue/");
 5
 6    List<File> trueFiles =
         Arrays.asList(Objects.requireNonNull(trueFolder.listFiles()));
 7
 8    return trueFiles.stream()
 9       .map(file -> DynamicTest.dynamicTest("Testing: '" +
            file.getName() + "'",
10       () -> assertTrue(MainParse.parseFile(file.getPath()))));
11 }
```

Listing 13.3 shows the implementation of a general *TestFactory* for tests that are expected to fail and throw an exception. This general implementation has been made because there are a number of reasons why a test should fail, whereas a true test should always get all the way through the compilation process. The general structure is equivalent to Listing 13.2, except for the fact that a specific class of expected exception can be given as a parameter, and it is asserted that the compilation throws the given exception.

Listing 13.3: Method to generate dynamic tests that are expected to fail and throw a specific exception

```java
// General test factory for false tests
private Stream<DynamicTest> expectedFalse(File filePath, Class
    expectedExceptionClass) {
   List<File> falseFiles =
      Arrays.asList(Objects.requireNonNull(filePath.listFiles()));

   return falseFiles.stream()
      .map(file -> DynamicTest.dynamicTest("Testing: '" +
         file.getName() + "'",
      () -> assertThrows(expectedExceptionClass, () ->
         MainParse.parseFile(file.getPath()))));
}
```

13.3 Extent of tests

The previous sections described how the unit tests and program tests are used in the validation of the correctness of the compiler. This section seeks to describe the extent to which these types of tests have been implemented, and their level of coverage in the compiler, which is a measurable way of determining the usefulness of the test suite. Measuring code coverage is often divided into three categories: Class coverage, method coverage, and line coverage which respectively is the number of classes the tests use, the number of methods that are used, and the number of runnable lines that are executed.

Some Java-files are auto-generated by JFlex and Cup, which in turn won't make sense to gain test coverage on. Furthermore, a bunch of custom exceptions has been created to ensure that when the compiler fails, it provides a useful exception and associated message. These exceptions have multiple constructors to allow both custom and standard error messages, some of which are not used and wouldn't make sense to test to increase coverage statistics. After excluding these files from the code coverage measurement, the percentage of classes, methods, and lines that are covered, can be found for the unit tests exclusively, the program tests exclusively, and lastly the collection of all available tests.

Table 13.1 gives an overview of the coverage percentages for each possible configuration. The project has a few lines that, if everything works correctly, should be unreachable which is the reason for some of the untested lines. The unit tests cover 81.7% of classes but only 69.4% of methods and 50.3% lines are covered in 312 unit tests, which is primarily caused by the different visitors which are very dependent on the previous phases and their resulting data, which is why it is difficult to configure a correct symbol table and the corresponding abstract syntax tree manually to run a specific test case. This is why the program tests are an integral part of the testing, as they supplement the unit tests and conversely.

The program tests cover a large percentage of the compiler which is expected to happen as they should cover all possible cases that the compiler is made to handle. To cover all cases, 312 different programs in the proposed language have been made, from which 203 contain main blocks which are tested again the ensure that the auto-generated code can run in java. This level of coverage does however not entail that all of this is correct, simply that the expected-true tests don't throw an exception, that they can run in java, and that the expected-false tests throw a specific exception in one of the visitors where it is expected to. As a result of this, the program tests are closely related to system tests as they each test a larger part of the compiler.

The previous parts have aimed to describe and account for the level of coverage gained by unit tests and program tests individually. They each cover different types of tests and when looking at the compiler in full, they are both necessary to ensure that the highest possible level of correctness is reached. This can also be measured by the coverage when running unit tests and program tests together, where all classes are covered and 94.4% of lines are covered in 827 tests.

	Program Tests	**Unit Tests**	**All Tests**
Class Coverage	99% (98/99)	81.7% (103/126)	100% (126/126)
Method Coverage	95.5% (386/404)	69.4% (384/553)	98% (542/553)
Line Coverage	92.7% (1,786/1,926)	50.3% (1,183/2,352)	94.4% (2,231/2,363)
Number of Tests	515	312	827

Table 13.1: Coverage of unit tests, program tests, and the collection of all tests

As previously described the development process was fairly test-driven, as lots of program tests were written prior to beginning development, and continuously added during development. The program tests were written to use language constructs, that were expected to work at some point, and thus became the testable goal when developing the compiler phases. A program test will thus keep failing until all of its used language constructs are implemented correctly.

Having the described level of test-coverage and an overview of the percentage of covered code helps increase trust in the quality of the software. This is primarily because there has been made an effort in order to design code using test-driven development and eliminate possible unwanted errors in the compiler.

13.4 Javadoc

Javadoc is a document generator for documenting Java source code, which is entirely contained in comments and thus does not affect the performance of the program. The format of these comments is an industry standard for documenting Java code. It allows the generation of an HTML page which displays an overview of all classes and methods. [32]

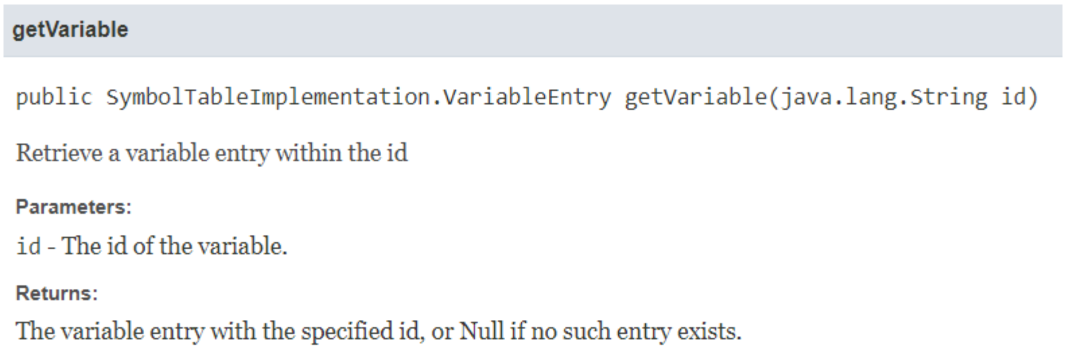
The general structure of a Javadoc-documented method is shown in Listing 13.4 where the '/**' on line 1 initializes the Javadoc comment, which is closed on line 5 with '*/'. The text on line 2 forms a general description of the method, the @param annotation indicates that the following description is of the parameter 'id', and the @return annotation indicates that the following description is of what the method returns. Additional annotations can be used to describe exceptions that are thrown, whether the code is deprecated, the version number of the code, and many more important qualities.

Listing 13.4: Javadoc documented method

```
1 /**
2  * Retrieve a variable entry within the id
3  * @param id The id of the variable.
4  * @return The variable entry with the specified id, or Null
       if no such entry exists.
5  */
6 public VariableEntry getVariable(String id) {
7     return this.scope.getEntry(id);
8 }
```

By following this way of commenting, a documentation page can be generated for the 'getVariable' method. This is shown in Figure 13.1, which is the result of the comment in Listing 13.4.

By consistently commenting classes and methods with this standard, a


getVariable

```
public SymbolTableImplementation.VariableEntry getVariable(java.lang.String id)
```

Retrieve a variable entry within the id

Parameters:

`id` - The id of the variable.

Returns:

The variable entry with the specified id, or Null if no such entry exists.

Figure 13.1: Javadocs generated HTML documentation based on Listing 13.4

comprehensive and detailed reference manual can be generated on top of getting a very approachable and maintainable code-base. The resulting Javadoc documentation page can be found as a zip-folder in the project-code in the accompanying documents.

2,429 lines of code in the project are comments and 8,665 lines are source code out of the 13,196 total lines in the 'src'-folder. Around 413 methods of 553 are covered with Javadoc, where the non-documented methods generally are either private or trivial.

The level of documentation of the source code further improves the point made in 'Extent of tests' [Section 13.3] about improved quality and trust in the software.

Summary

This chapter elaborated on software quality and showed how testing could help implement easily maintainable code. Furthermore, it was discussed how different components interacting in the implementation can be covered by tests. Lastly, Javadoc has been used to create documentation of the functionality, which make it easier to understand.

With Javadoc and the software quality in general explained and accounted for, it's possible to reflect upon the design, implementation, and quality of the project in the reflection part.

Part V

Reflection

14. User Evaluation

This chapter will focus on evaluating the project language. The goal is to figure out where there might be some inconsistencies or vagueness. The evaluations will be done with user evaluations where three fourth semester computer science students are assigned some tasks to complete and to help evaluate the language.

User evaluations have been performed, where three students individually were assigned a set of exercises to complete. Attached to the exercises, was a small reference manual describing the basic syntax and constructs of the language. These exercises asked the users to build and construct blocks of different levels of content based on a graphical representation of the resulting model. The users were asked to think aloud and specifically say when they felt that they had completed an exercise. The materials provided in the evaluation such as exercises, syntax reference manual, and follow up questions can be found in Appendix E.

The following section will go through the notable results of the user evaluations.

14.1 Results

This section will introduce and discuss the results of the evaluation of the project language. Table 14.1 outlines the time it took for each test person to finish each exercise. The second column shows the time to complete each exercise by a reference test person. The reference person wasn't previously exposed to the exercises, but is a creator of the language, and consequently a skilled user of the language.

There was no time limit to complete the exercise. The time was measured, simply to get an estimate of how easily the users were able to use the language. It's noted that time used per exercise is not a completely accurate metric. Variables such as typing speed could affect the time used, but not reflect the test persons understanding of the language.

Exercise	Reference	Test Person 1	Test Person 2	Test Person 3
1.1	*0:12*	0:24	0:20	0:30
1.2	*0:09*	0:35	1:00	0:55
1.3	*0:14*	0:50	2:30	1:00
1.4	*0:20*	1:40	1:13	1:55
1.5	*0:30*	1:10	2:30	2:30
1.6	*0:30*	1:30	1:30	2:20
2.1	*0:37*	0:45	1:20	5:00
2.2	*0:42*	2:15	1:00	5:00
2.3	*0:34*	6:00	4:00	2:00

Table 14.1: Time to complete exercises

The users used their time quite differently. Some reasons for this could be that some users understood the tasks better than others, while some users simply were more intrigued by the experiment and used more time to get it right. The last and most difficult task had the users reflecting on their prior tasks to ensure they were completed correctly. In this task, they had to connect all instantiated elements correctly, but this caused a substantial time-consumption. Something that consumed time, was that some users wanted to make sure their final results were satisfying to them.

The results show, that on average the students used 25 seconds on completing the first exercise. The average time used per exercise was expected to grow in correlation with progression since the difficulty of each exercise also increased. This is shown as the average time for the last and most difficult exercise had the test persons use an average of 4 minutes.

During the experiment, some level of assistance was given in order to help the users progress. If the user were to ask a question about the language, an observer would answer that question with a low level of detail in order to let the user figure out the details themselves using the provided resources. In cases where the user felt their answer was correct, but indeed had mistakes, the observer would not correct them or inform them of this, until the end of the experiment.

Generally, the users felt that the connections were straight forward and made sense. Appendix E shows the notes for each of the exercises and final questions. The blueprint felt like a constructor known from object-oriented languages. Most users didn't use the short-hand connection notation with operations but did appreciate the dot-notation since that's well known from object-oriented languages. A point of confusion for some users was the predefined operations. The confusion might arise from the fact that they didn’t see them as blocks that the compiler has already implemented. Without understanding this, it would consequently also be difficult to understand that they could build them without context and that they had connectable channels predefined as well. This could have been improved by giving a more detailed introductory description of the language and make the operations in the illustration look like blocks.

Some elements that could affect the outcome of these evaluations consist of the quality of the reference manual appendix, the overall explanation given of the language and the difference in illustrations used in the tasks.

Summary

Even though only a few users were used to conduct an evaluation, the results of those evaluations gave some insight into which elements of the project language could cause confusion and whether the language actually has the level of writability that is desirable. All in all, the evaluations gave promising results, that showed students would be able to learn to construct their own networks using the proposed dataflow oriented programming language.

15. Discussion

This chapter will highlight to what degree key features have been implemented correctly and which challenges may have come up to prevent further development. Furthermore, it will catch up on the requirements for this project in order to evaluate the degree to which they have been satisfied, and discuss the user evaluations introduced in the previous chapter.

15.1 Key functionality

The project language was developed with an intention to exclude the need to define the mathematic operations in machine learning development. This should make students or enthusiasts have an easier time learning and understanding machine learning models and structures, by making a correlation between how machine learning models are visualized through directed graphs, and how they are implemented. This was partly achieved by developing a language in the dataflow oriented paradigm.

In the Initial Problem [Section 2.1], three key issues with the existing solutions, were identified:

- Languages being used for machine learning were designed for too broad a range of applications
- Libraries are restricted by the predefined language constructs and syntax.
- Implementations using libraries in general-purpose languages won't accurately reflect the mental model of the programmer.

Starting off with the first point, it's argued that the proposed language definitely has solved this issue. The proposed language is designed for a niche purpose, namely as a structural language to build blocks and connect them to form Artificial Neural Network models. As mentioned in Code Generation [Section 9.3], the proposed language compiles to a Java library. Therefore, one could argue that the language is bound to the constructs of the library, but seen from a syntax perspective, the language is independent of the library. By this, it's argued that the second issue has been solved with a syntax-focus in mind. As mentioned earlier, a user evaluation was held towards the later stages of development. Here the following question was asked: *Did you face any major issues, converting the illustrations given in the exercises, to code?* All three test people answered this question roughly the same way [Appendix E]. The way the language is structured, made it reflect the illustrations quite well, though sometimes with a little assistance. It is only expected that the test people needed a small amount of assistance, as they were only given a small reference manual with the needed syntax [Appendix E], and a small amount of time to understand, and complete the exercises. This is used as an argument towards solving the third and last key issue. The user evaluations will be discussed further in Section 15.4.

In order to create a functioning programming language, it's important to develop a functioning translator to go with it. Here it was chosen to develop a compiler.

All phases of the compiler have been implemented. The compiler works as intended and it's possible to translate a program in the proposed language to an equivalent runnable program in Java.

15.2 Challenges

During development many challenges came up, which has changed the expected outcome of the programming language and the way the compiler works. This section will also cover some of those features that wasn't made within the first iterations and therefore has been left out

A challenge that has created some difficulties throughout the project is how the compiler should work with the project language and compilation to Java. The current implementation compiles to a library in Java. The current structure of the compiler would not be suitable if the language were to be Turing complete. Adding conditions, loops, branches and other popular programming concepts could benefit some aspects of the programming language while limiting the user experience of others, because the relatively small size of the language helps the general goal of readability and writability. Therefore, finding the right compilation process for this current iteration has been a recurring challenge.

15.3 Requirement evaluation

When working on a project it's important to work towards fulfilling the stated requirements. When doing this, some requirements are easily fulfilled, while some are harder due to known or later discovered difficulties. With that in mind, this section will give an overview of the requirement evaluation and to what degree those requirements have been fulfilled.

All of the must-have functionalities have been fulfilled, which ensures that the minimum viable product is in fact, viable, which means that it can be used and tested upon for further evaluation. Completing the must-have tasks allows for development of features which are stated as could-have features. A good requirement, is one that is testable, and as mentioned in Chapter 13, several tests were written to ensure a certain quality of the compiler. Some of these test ensure that certain requirements have been fulfilled, and therefore, it's quite safe to say which requirements have been met, through software tests.

One feature has been addressed partly, since the original requirement stated that blocks should be able to connect both fully and partly and that certain exceptions or warnings should be given in special cases [L15]. Here it was decided that block should be fully connected or the semantic phase of the compiler would throw a *SemanticProblemException*.

To summarize, a large amount of requirements were set at the very early stages of development, to allow an iterative approach. All must-have requirements, and some should-have requirements, have been met during the first iteration of development, which was tested with program tests. Should the language be developed further at some point, there is still a set of less critical requirements that can be met, to further enhance the language.

15.4 User Evaluation

This section will discuss whether future iterations of the project language will include changes based on the results of the user evaluations. As mentioned in Chapter 14, the users found the operations to be the most unfamiliar and confusing concept of the language. That might have been due to the circumstance of the evaluations, but there might be areas that are open to change.

Operations are less informative in the sense, that they are predetermined and created beforehand, without the user having to create them, which to some, might seem odd. This is not seen as something, that needs to be changed, as it is only natural that some aspects of the language simply need to be understood beforehand. The context that this programming language is going to be used in is for educational use, and in those cases, the students should be able to quickly implement a functioning program, without having to go in detail with their operations. Therefore it seems most fitting, that operations stay the way they are.

Summary

An evaluation of the project's challenges, and the state in which the programming language currently exist, has been completed in this discussion. From this, it's possible to conclude on the overall discoveries made in this report.

16. Conclusion

To conclude upon the discoveries and reflection made in this project, all sub-conclusions will benefit as arguments for how the development has helped to solve the overall problem statement.

The initial problems that were identified during the problem analysis, were that languages used for machine learning are used for too broad a range of applications, and that machine learning libraries are restricted by the syntax of their language. To overcome these problems, a language dedicated to building artificial neural network models, with syntax specifically supporting this, is proposed. Through the stakeholder analysis, the primary purpose of the proposed language is identified as an educational language. These discoveries appear from the problem statement:

How can a programming language be designed and its appurtenant compiler implemented, to help increase the educational potential, of working with Artificial Neural Networks?

To answer this quite extensive question, three sub-questions were established:

- Which syntactic, contextual, and semantic language constructs will allow a high level of readability and writability?
- Which information should the individual compiler phases accept and output to allow translation from the proposed language to Java code that can be further compiled into Java byte-code and machine code - and how should such phases be implemented?
- How can the educational potential and language criteria be evaluated?

To answer the first sub-question, the proposed language was compared to an existing state-of-the-art machine learning library named Keras, which served as the basis for an analysis of the core language criteria.

Based on this, as well as the requirement specification, the specification of the language design was made to facilitate readability and writability. The aforementioned requirement was fulfilled by making the syntax convey what the purpose of the individual specific language construct is and keeping the code model close to the mental model of the programmer. The result of this is a language with few operators, which makes it possible to be expressive, without writing too much code because of the simple syntax and that much of the underlying math has been hidden from the programmer. Even though Keras was made the point of reference, it is still a vastly different tool in relation to the proposed language, and in many ways better for the language's main demographic. The comparison was made to highlight potential points of improvement, which in specific was the syntax and through that the structure of the code. These areas were found to potentially have been improved from the subset of Keras that is compared.

The second sub-question assumes that a language has been designed and should be translated to machine code to allow running programs written in the language. The question is answered throughout the translator design part and the implementation part, where an AST and symbol table is created, and each phase traverses the AST and makes inquiries in the symbol table. When the validity of the program has been confirmed, the code generation phase generates corresponding Java code, and thus allows running the program. The correctness and quality of the aforementioned implementation are assured through unit testing, program testing, and documentation.

To answer the third sub-question a user-evaluation of the language was held. Here, a number of tasks were put forward to evaluate the usage of the core language constructs. The results were generally positive as all 3 test-users got through the exercises with minimal guidance, in a relatively short amount of time, and few compile errors which would likely have been fixed if they were given the corresponding error messages. As appears from the follow-up questions in the notes in Appendix E none of the test people had worked with machine learning previously but they all managed to complete nearly all the exercises in a relatively short time. They were also generally positive towards the language, which suggests that the simple syntax and the small size of the language have some educational potential.

A programming language was thus designed, and its appurtenant compiler implemented. The language was designed with educational potential in mind. The result of this, was a dataflow-oriented language, designed to implement Artificial Neural Networks. Its syntax is meant to increase the readability and writability for people wanting to learn about machine learning, such as students or enthusiasts. This was accomplished by eliminating general-purpose features, and thus, aiming the use of the language towards fast-paced development of Neural Networks. The developed compiler compiles the language to Java code, and the compiled code can be handled as a library, which is the access point for the compiled code.

Bibliography

[1] Enterprise Management 360. *Top 10 companies using Machine Learning*. URL: https://www.em360tech.com/tech-news/top-ten/top-10-companies-using-machine-learning/. September 25, 2018.

[2] Macy Bayern. *Why machine learning will see explosive growth over the next 2 years*. URL: https://www.techrepublic.com/article/why-machine-learning-will-see-explosive-growth-over-the-next-2-years/. September 18, 2018.

[3] Piotr Migdał. *Simple diagrams of convoluted neural networks*. URL: https://medium.com/inbrowserai/simple-diagrams-of-convoluted-neural-networks-39c097d2925b. Sep 15, 2018.

[4] Christina Voskoglou. *What is the best programming language for Machine Learning?* URL: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7/. May 5, 2017.

[5] Erik Brynjolfsson Ryan Calo Oren Etzioni Greg Hager Julia Hirschberg Shivaram Kalyanakrishnan Ece Kamar Sarit Kraus Kevin Leyton-Brown David Parkes William Press AnnaLee Saxenian Julie Shah Milind Tambe Peter Stone Rodney Brooks and Astro Teller. *Artificial Intelligence and Life in 2030." One Hundred Year Study on Artificial Intelligence: Report of the 2015-2016 Study Panel*. URL: https://ai100.stanford.edu/2016-report/section-i-what-artificial-intelligence/ai-research-trends/. September 6, 2016.

[6] Stacy Stanford. *Artificial Intelligence: Salaries Heading Skyward*. URL: https://medium.com/mlmemoirs/artificial-intelligence-salaries-heading-skyward-e41b2a7bba7d. Aug 29, 2018.

[7] Cade Metz. *Tech Giants Are Paying Huge Salaries for Scarce A.I. Talent*. URL: https://www.nytimes.com/2017/10/22/technology/artificial-intelligence-experts-salaries.html. Oct 22, 2017.

[8] Adobe Inc. Econsultancy. *Digital Intelligence Briefing: 2018 Digital Trends*. February, 2018.

[9] Wikipedia. *R (programming language)*. URL:https://en.wikipedia.org/wiki/R_(programming<_language). February 4, 2019.

[10] Wikipedia. *Python (programming language)*. URL:https://en.wikipedia.org/wiki/Python_(programming_language). February 4, 2019.

[11] Prince Patel. *Why Python is the most popular language used for Machine Learning*. URL:https://medium.com/@UdacityINDIA/why-use-python-for-machine-learning-e4b0b4457a77. March 8, 2018.

[12] Will Koehrsen. *Modeling: Teaching a Machine Learning Algorithm to Deliver Business Value*. URL:https://towardsdatascience.com/modeling-teaching-a-machine-learning-algorithm-to-deliver-business-value-ad0205ca4c86. November 15, 2018.

[13] Christina Voskoglou. *What is the best programming language for Machine Learning?* URL: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7. May 5, 2017.

[14] John Olsson Mette Lindegaard. *Power i projekter of porteføjler*. URL:https://www.moodle.aau.dk/pluginfile.php/1005168/mod_folder/content/0/TMS_kap4_ppp.pdf?forcedownload=1. 2005.

[15] Jeff Hale. *Deep Learning Framework Power Scores 2018*. URL:https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a. September 20, 2018.

[16] Matt Carkci. *Dataflow and Reactive Programming Systems*. Leanpub, 2014.

[17] Tiago Boldt Sousa. "Dataflow Programming Concept, Languages and Applications". In: *Doctoral Symposium on Informatics Engineering* (2012), pp. 323–334.

[18] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 2016.

[19] P. Nielsen J. Stage L. Mathiassen A. Munk-Madsen. *Object Oriented Analysis and Design*. Metodica, 2000.

[20] Ken Schwaber. "SCRUM Development Process". In: *Business Object Design and Implementation*. Ed. by Jeff Sutherland et al. London: Springer London, 1997, pp. 120–125.

[21] Richard J. LeBlanc. Jr. Charles N. Fisher Ron K. Cytron. *Crafting A Compiler*. Pearson, 2009.

[22] Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010.

[23] Wikipedia. *Scope (Computer Science)*. URL:https://en.wikipedia.org/wiki/Scope_(computer_science). February 3, 2019.

[24] JFlex. *JFlex - What is it?* URL: https://www.jflex.de/. April 2, 2019.

[25] javatpoint.com. *Parse tree and Syntax tree*. URL: https://www.javatpoint.com/parse-tree-and-syntax-tree. April 9, 2019.

[26] Tutorialspoint. *Compiler Design - Code Generation*. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.htm. March 18, 2019.

[27] Ankit Kumar Singh. *Compiler Design - Symbol table*. URL: https://www.geeksforgeeks.org/symbol-table-compiler/. April 2, 2019.

[28] Scott Hudson. *CUP 0.11b*. URL: http://www2.cs.tum.edu/projects/cup/. April 4, 2019.

[29] Magnus Madsen. *Software Testing*. URL:https://www.moodle.aau.dk/pluginfile.php/1173653/mod_resource/content/1/Lecture%201a%20-%20Software%20Testing.pdf. February 10, 2018.

[30] Noela Jemutai Kipyegen and William P. K. Korir. "Importance of Software Documentation". In: *IJCSI International Journal of Computer Science* 10.1 (2013), pp. 223–228. ISSN: 1694-0784.

[31] Panikaj JournalDev. *JUnit 5 Dynamic Tests – @TestFactory, DynamicTest*. URL:https://www.journaldev.com/21715/junit-dynamic-tests-testfactory-dynamictest. 2019.

[32] Wikipedia. *Javadoc*. URL:https://en.wikipedia.org/wiki/Javadoc. May 4, 2019.

Part VI

Appendices

A. Tokens and regular expressions

Keywords	Regular expression
mychannelin	”$mychannel : in$”
mychannelout	”$mychannel : out$”
channelin	”$channel : in$”
channelout	”$channel : out$”
block	”$block$”
operation	”$operation$”
source	”$source$”
connection	”$- >$”
build	”$build$”
draw	”$draw$”
blueprint	”$blueprint$”
procedure	”$procedure$”
this	”$this$”

Symbols	Regular expression
comma	”,”
dot	”.”
assign	” = ”
lparen	”(”
rparen	”)”
lcurly	”{”
rcurly	”}”
semi	”;”

B. Full Production Rules

```
# Program
<Prog>
   → <Blocks>

# Block
<Blocks>
   → <Block>
   | <Block> <Blocks>

<Block>
 → block id lcurly <Definition> rcurly

<Definition>
 → <ChannelDeclarations> <Blueprint> <Procedures>

# General
<Type>
 → <BuildType>
 | <BlueType>
 | <Channel>
 | <MyChannel>

<BuildType>
 → block
 | operation
 | source

# Channels
<ChannelDeclarations>
 → <InChannelDeclaration> <OutChannelDeclaration>
    <InOutChannelDeclaration>

<InChannelDeclaration>
   → channelin id semicolon <InChannelDeclaration>
   |  λ

<OutChannelDeclaration>
   → channelout id semicolon

<InOutChannelDeclaration>
   → <Channel> id semicolon <InOutChannelDeclaration>
   |  λ

<MyChannel>
 → mychannelin
 | mychannelout

<Channel>
```

```
 → channelin
 | channelout

# Blueprint
<Blueprint>
 → blueprint lpar <BlueParams> rpar lcurly <Content> rcurly

<BlueParams>
 → <BlueParam>
 |  λ

<BlueParam>
 → <BlueType> id <BlueExtParams>

<BlueExtParams>
 → comma <BlueParam>
 |  λ

<BlueType>
 → blueprint
 | size

# Content
<Content>
 → <Statement> semicolon <Content>
 |  λ

<Statement>
 → <Declaration>
 | id <OptIdStatement>
 | lpar <OptLParStatement>
 | this dot id <OptThisStatement>
 | <BuildDeclaration> <Chain>
 |  λ

<ProcedureCall>
 → lpar <ElementListStart> rpar

# Option select for Statements
<OptLParStatement>
 → lpar <Element> rpar <GroupCon>
 | <Element> <ParCloseStatement>

<ParCloseStatement>
 → <ParChain>
 | <GroupCon>

<OptIdStatement>
 → <DotId> <Chain>
 | <ProcedureCall>
 | assign <ElementPar> <DeclStopOrStay>

<OptThisStatement>
 → <ProcedureCall>
 | <Chain>

# Collective Statements
<Chain>
 → connection <NodeElementPar> <ChainExt>

<ParChain>
 → rpar <Chain>
```

```
<ChainExt>
 → <Chain>
 |   λ

<GroupCon>
 → <ElementList> rpar connection <ElementPar> <ChainExt>

# Elements
<Element>
 → <NodeElement>
 | <NonNodeElement>
 | <NonNodeDeclaration>

<NodeElement>
 → <NodeDeclaration>
 | id <DotId>
 | this dot id
 | id assign <NodeSelectors>
 | <BuildDeclaration>

<NonNodeElement>
   → draw id
   | <SizeConst>

<NodeSelectors>
 → id dot id
 | this dot id
 | <BuildDeclaration>
 | id

<BuildElement>
 → <BuildDeclaration>
 | id
 | lpar <BuildElementPar>

<BuildElementPar>
 → <BuildDeclaration> rpar
 | id rpar

<DotId>
 → dot id
 |   λ

<ElementPar>
 → lpar <Element> rpar
 | <Element>

<NodeElementPar>
   → lpar <NodeElement> rpar
   | <NodeElement>

<SizePar>
   → lpar <SizeConst> rpar
   | <SizeConst>

<SizeConst>
   → lsqr intconst comma intconst rsqr

<ChannelElementsPar>
   → lpar <ChannelElement> rpar
   | <ChannelElement>
```

```
<ChannelElement>
   → this dot id
   | id <DotId>

<IdPar>
   → lpar id rpar
   | id

# Declaration Element
<NodeDeclaration>
   → <BuildType> id <BuildAssign>

<NonNodeDeclaration>
   → blueprint id <BlueprintAssign>
   | size id <SizeAssign>
   | <Channel> id <ChannelAssign>

<Declaration>
 → blueprint id <BlueprintAssignOpt>
 | size id <SizeAssignOpt>
 | <Channel> id <ChannelAssignOpt>
 | <BuildType> id <BuildDecla>

<BuildDecla>
   → <BuildAssign> <DeclStopOrStay>
   |  λ

<BlueprintAssign>
   → assign draw <IdPar>

<SizeAssign>
   → assign <IdPar>

<ChannelAssign>
   → assign <ChannelElementsPar>

<BuildAssign>
   → assign <BuildElement>

<BlueprintAssignOpt>
   → <BlueprintAssign>
   |  λ

<SizeAssignOpt>
   → <SizeAssign>
   |  λ

<ChannelAssignOpt>
   → <ChannelAssign>
   |  λ

<BuildDeclaration>
 → build id lpar <ElementListStart> rpar

<DeclStopOrStay>
 → <Chain>
 |  λ

# ElementList
<ElementListStart>
 → <ElementPar> <ElementListExt>
 |  λ
```

```
<ElementList>
 → <ElementExt> <ElementListExt>

<ElementListExt>
 → <ElementList>
 |   λ

<ElementExt>
 → comma <ElementPar>

# Procedures
<Procedures>
 → procedure lpar <P_Params> rpar lcurly <Content> rcurly <Procedures>
 |   λ

<P_Params>
 → <Type> id <P_ExtParams>
 |   λ

<P_ExtParams>
 → comma <P_Params>
 |   λ
```

C. MoSCoW

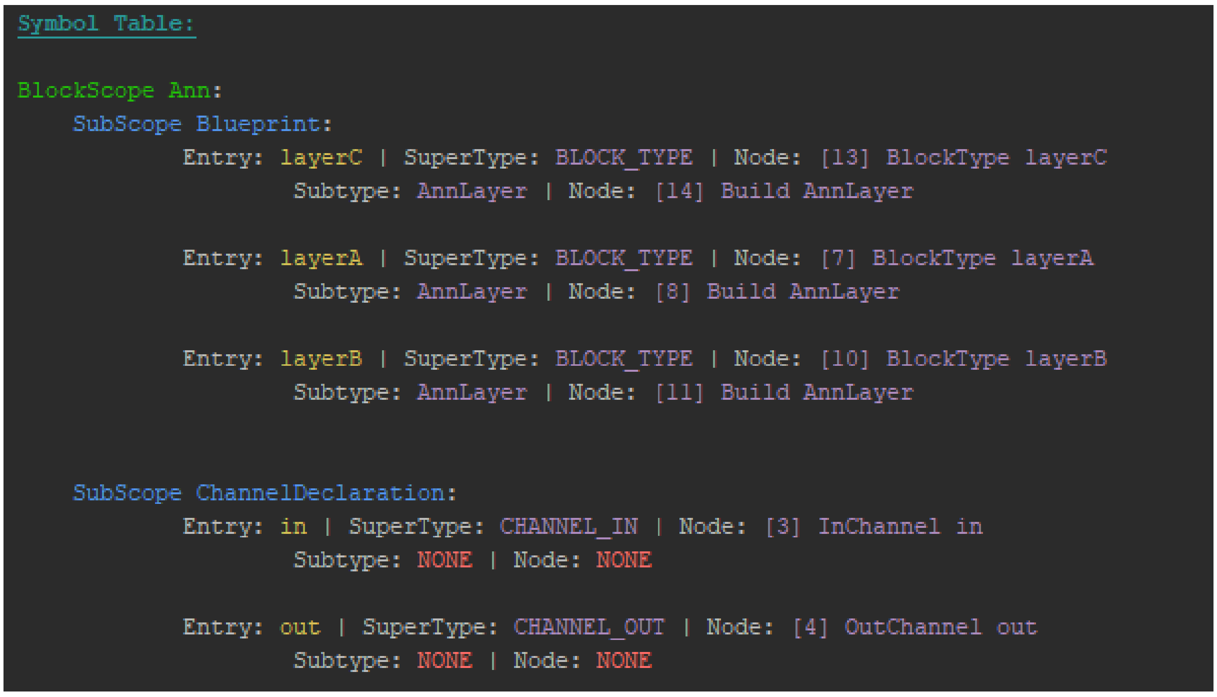
The requirements that in the MoSCoW analysis were valued lower than "Must have".

- **L6 No useless non-terminals** — Should have
  Unreachable production rules, that can crop up when changes are made to the production rules, only clutter up the specification. They don't cause harm, but should be removed whenever they are found.
- **L13 Nestable block structure** — Should have
  Nestable blocks will allow for working within blocks with other blocks that represent anything from complex operations to networks of networks.
- **C2 Give useful error messages** — Should have
  Not only telling the user that they made an error, but where it is and what is wrong, is the highest priority for the compiler beyond the minimum viable product.
- **L18 Block Procedures** — Should have
  Procedures allow for significant improvement of readability and writability of code where sequences of actions are repeated.
- **I2 Connection notation as arrow** — Should have
  Using arrows in code as the syntax to represent arrows on a model matches Requirement **??** as well.
- **I3 Strict sizes** — Should have
  For straightforward neural networks, the sizes of the signals within the network can be inferred from the size of the input and output. However, needing to specify the sizes ensures readability and understanding of how the networks work, which is important for the target userbase.
- **L9 Self-descriptive Keywords** — Should have
  For ease of use and understanding, having the keywords of the language convey what they represent makes a big difference.

- **L19 Limited Access to simple datatypes** — Should have
  The purpose of the language is to build chains of operations to handle a dataflow, and not for imperative programming.
- **I1 Gate renamed to Channel** — Should have
  Renaming the keyword "gate" to "channel" to avoid clashing with the existing concept named "gate" in machine learning is nothing vital for the language to serve its purpose, but it is an improvement to be made without meaningful drawbacks.
- **I5 Know the target group** — Should have
  To make a satisfactory product, it is quite important to know what the target group needs out of it.
- **L12 Recurrent blocks** — Could have
  Allowing blocks to be connected to themselves would allow for making recurrent neural networks, but the implementation of backpropagation through these is significantly more complex.
- **L15 Fully and partly connected blocks** — Could have
  Giving the user options for what they want to connect or not, is not absolutely necessary to achieving any functionality, but can have a significant impact on ease use.
- **C3 Graphical modelling of networks; Compiler Option** — Could have
  Showing the user a model of their implementation is a smooth way for the user to verify if they implemented what they were going for, but not central for the functionality of the language.
- **L7 Intuitive Language Comprehension** — Could have
  Having the final code be immediately intuitive would show it to have a high degree of readability. There is however more focus on having the language be simple to learn from the core conceit.
- **L10 Optional Indenting** — Could have
  Indenting in the language should ideally be entirely up to the users sensibilities. But if something gets in the way of that, it may be sacrificed.
- **L20 Procedure calling other procedures** — Could have
  Calling procedures from within procedures allows for further abstraction, but if it proves problematic, it is not something vital.
- **L11 Custom operations** — Won't have
  Allowing the users to define their own operations would allow for

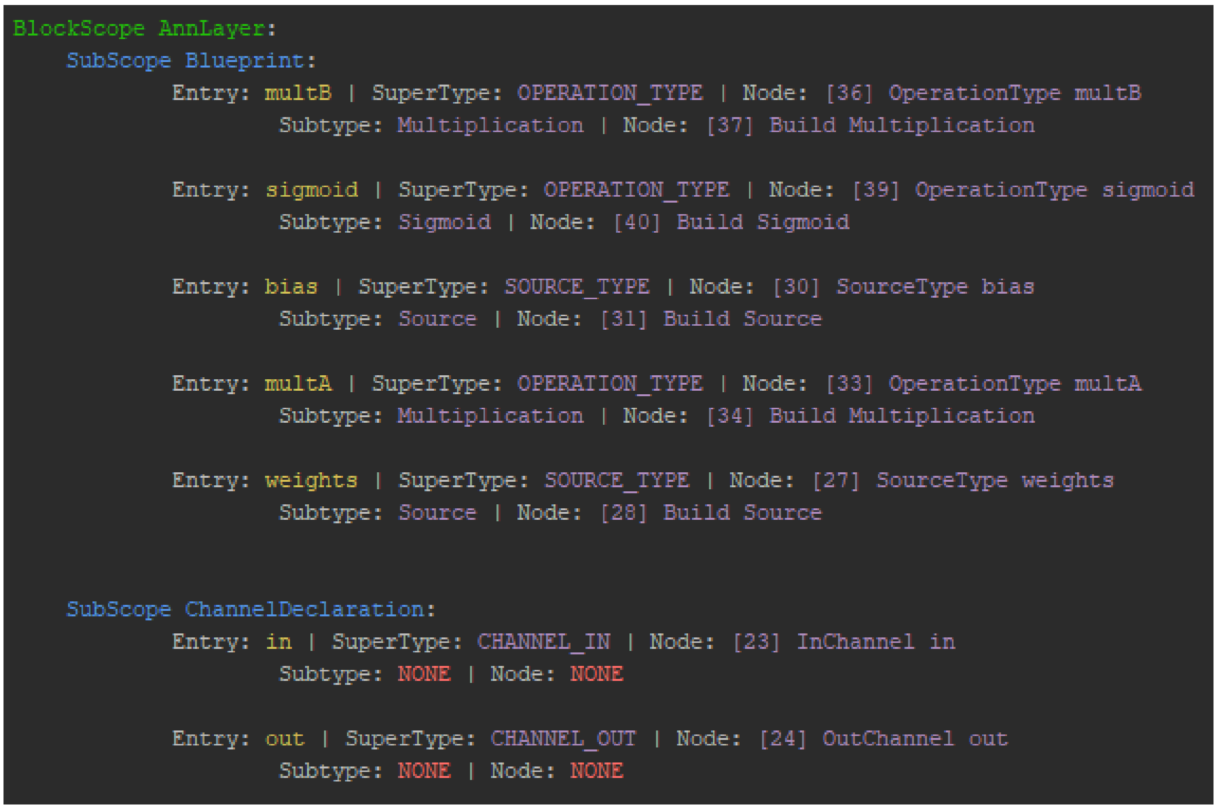
broader use cases for the language, but require the user to implement forward- and backpropagation themselves, which would require several additional concepts in the language, that would be largely disconnected from how the rest of the language. Because of this, custom operations is not deemed to be within the scope of the project.

D. Symbol Table



```
Symbol Table:

BlockScope Ann:
    SubScope Blueprint:
            Entry: layerC | SuperType: BLOCK_TYPE | Node: [13] BlockType layerC
                    Subtype: AnnLayer | Node: [14] Build AnnLayer

            Entry: layerA | SuperType: BLOCK_TYPE | Node: [7] BlockType layerA
                    Subtype: AnnLayer | Node: [8] Build AnnLayer

            Entry: layerB | SuperType: BLOCK_TYPE | Node: [10] BlockType layerB
                    Subtype: AnnLayer | Node: [11] Build AnnLayer


    SubScope ChannelDeclaration:
            Entry: in | SuperType: CHANNEL_IN | Node: [3] InChannel in
                    Subtype: NONE | Node: NONE

            Entry: out | SuperType: CHANNEL_OUT | Node: [4] OutChannel out
                    Subtype: NONE | Node: NONE
```

Figure D.1: Console Symbol Table Example Print



```
BlockScope AnnLayer:
    SubScope Blueprint:
            Entry: multB | SuperType: OPERATION_TYPE | Node: [36] OperationType multB
                    Subtype: Multiplication | Node: [37] Build Multiplication

            Entry: sigmoid | SuperType: OPERATION_TYPE | Node: [39] OperationType sigmoid
                    Subtype: Sigmoid | Node: [40] Build Sigmoid

            Entry: bias | SuperType: SOURCE_TYPE | Node: [30] SourceType bias
                    Subtype: Source | Node: [31] Build Source

            Entry: multA | SuperType: OPERATION_TYPE | Node: [33] OperationType multA
                    Subtype: Multiplication | Node: [34] Build Multiplication

            Entry: weights | SuperType: SOURCE_TYPE | Node: [27] SourceType weights
                    Subtype: Source | Node: [28] Build Source


    SubScope ChannelDeclaration:
            Entry: in | SuperType: CHANNEL_IN | Node: [23] InChannel in
                    Subtype: NONE | Node: NONE

            Entry: out | SuperType: CHANNEL_OUT | Node: [24] OutChannel out
                    Subtype: NONE | Node: NONE
```

Figure D.2: Console Symbol Table Example Print

E. User Evaluations

Exercise 1

The goal is to build a block with one input and one output. Inside the block is three blocks (layers), each also with one input and one output. The illustration below shows a visual representation of the block we wish to build (The light dots represent input channels, and the dark dots represent output channels):

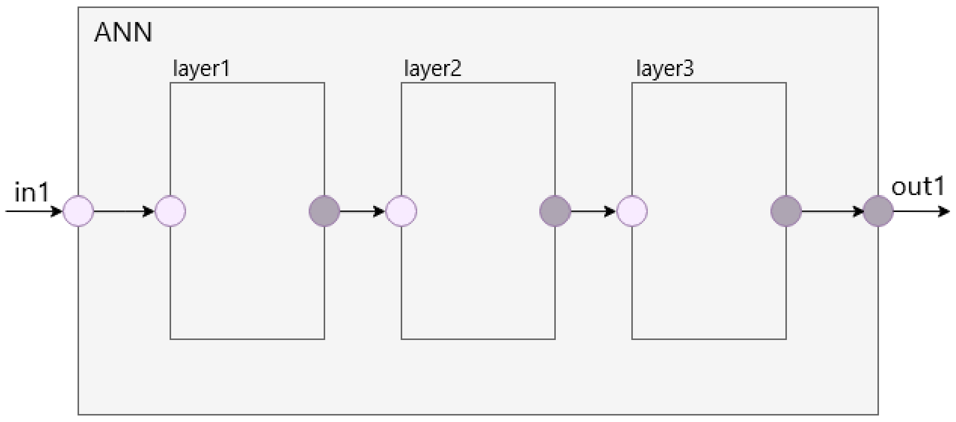


Figure E.1: Ann Model - Exercise 1

First we need to declare one of the inner blocks, called AnnLayer. This block is shown in the figure below (This block is the same as the blocks: layer1, layer2, layer3 in the above illustration):

Exercise 1.1

Declare a block, and give it the id: AnnLayer

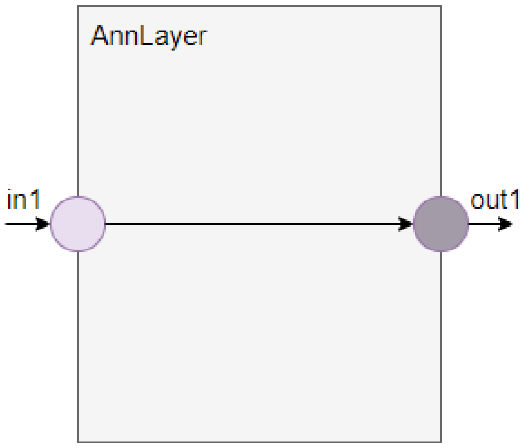


Figure E.2: Ann Layer Model - Exercise 1

Exercise 1.2

Declare one input and one output channel for the block, name them whatever you find suitable

Exercise 1.3

Declare the blueprint for your block, and in that blueprint connect the input channel to the output channel

We have now created a layer, which feeds data directly through the block from the input channel to the output channel. We now wish to create the block from the start of the exercise

Exercise 1.4

Declare a new block, called ANN, and give it one input and one output channel

Exercise 1.5

Declare the block's blueprint. In that blueprint build 3 AnnLayer's, and name them something appropriate

Exercise 1.6

In the blueprint, after the build declarations, make a connection, that feed data from the block's input, through all three layers, to the block's output

Exercise 2

Now that we have a basic understanding of channel declarations, blueprints, and connections, we can create a more interesting block, as well as introduce operations. The goal with this exercise is to create a block with inputs: One matrix A, and one matrix B. The block should add the matrices together, and then multiply the result with the input matrix B. A visual representation of this block is shown below:

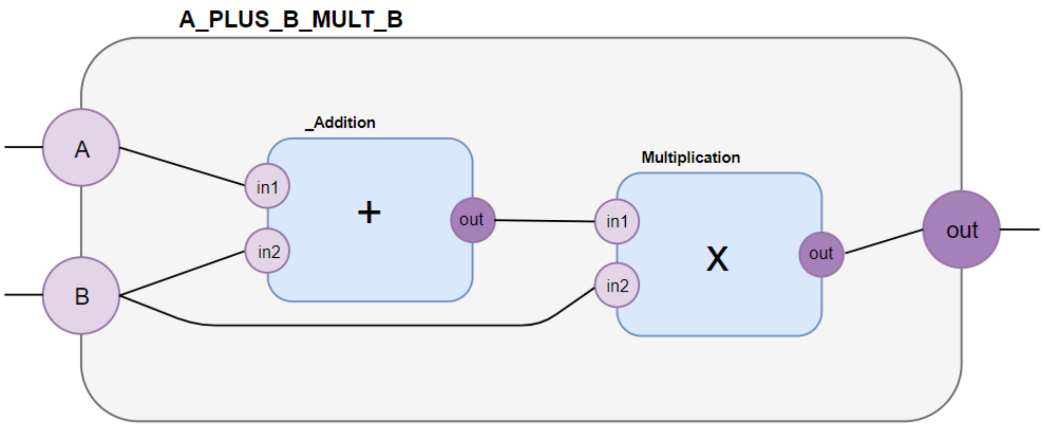


Figure E.3: A plus B mult B model - Exercise 2

Exercise 2.1

Declare the block, as well as its in- and output channels

Exercise 2.2

Create a blueprint and declare and build the necessary operations (a list of predefined operations can be found in the appendix)

Exercise 2.3

Connect the elements declared, to create the block shown in the illustration above (the names of the in- and output channels of operations, are as they appear in the illustration)

Syntax reference manual

Listing E.1: Block Example

```
block blockId {
 // MyChannel declarations

 // Blueprint

 // Optional procedure scopes
}
```

Listing E.2: Channel Declaration Example

```
mychannel:in id;
mychannel:out id;
```

Listing E.3: Scope Example

```
blueprint() {

}

procedure() {

}
```

Listing E.4: Variable Declaration Example

```
block id = build blockId();
operation id = build operationId();
source id = build Source();
```

Listing E.5: Connection Example

```
this.id -> this.id;
id -> id;
id -> id -> id;
(id, id) -> id
```

Listing E.6: Comment Example

```
// Comment
/*
 Comment
*/
```

Listing E.7: Predefined Operations

```
_Addition  // Unit-wise addition
_Subtraction // Unit-wise subtration
Multiplication // Matrix-wise multiplication
```

Notes

Test Person 1

Exercise 2.2: "Skal jeg lave et blueprint til hver operation? Nej jeg burde kunne lave det i én"
Exercise 2.3: Denne opgave forvirrer, er lidt i tvivl om det kan gøres i ét blueprint. Test Supervisor forklarer. Det at operations er blocks, var ikke så klart. Det giver godt mening med en lille forklaring

Question 1: Har du før arbejdet med maskinlæring?
Answer: Nej

Question 2: Følte du dig i stand til at løse opgaverne?
Answer: Ja, men connection eksemplerne forvirrede lidt

Question 3: Hvilke koncepter bed du mærke i?
Answer: God syntax, connections giver godt mening, dot-notaion er god, da den er som man kender den. Blueprint giver også mening, jeg tænker på den som en constructor i Java. Selve opbygning af programmer giver god mening.

Question 4: Var der nogle besværligheder ved at omdanne illustrationerne til kode?
Answer: Illustrationerne til de to opgaver er lavet forskellige, hvilket kunne forvirre lidt.

Test Person 2

Exercise 1.3: "Skal blueprint være indenfor blokken?"
Exercise 1.5: Det var ikke så ligetil at bygge tre blokke inde i en blok
Exercise 1.6: Connecte de tre layers fint, men de blev ikke connected til den ydre blok
Exercise 2.1: Kun ét input blev lavet
Exercise 2.3: Opgaven var for svær, og var ikke færdigjort

Question 1: Har du før arbejdet med maskinlæring?
Answer: Nej

Question 2: Følte du dig i stand til at løse opgaverne?
Answer: I store træk, ja

Question 3: Hvilke koncepter bed du mærke i?
Answer: Connections var smarte. Nemt at forstå med lidt forklaring

Question 4: Var der nogle besværligheder ved at omdanne illustrationerne til kode?
Answer: Første opgave var nem, anden opgave var lidt svær at forstå

Test Person 3

Exercise 2.2: Der er lidt forvirring omkring hvordan operations skal bruges. "Jeg går ud fra at man builder en operation på samme måde som man builder en blok"
Exercise 2.3: Opgaven var for svær, og blev ikke lavet færdig

Question 1: Har du før arbejdet med maskinlæring?
Answer: Nej

Question 2: Følte du dig i stand til at løse opgaverne?
Answer: Ja, id'er forvirrede lidt

Question 3: Hvilke koncepter bed du mærke i?
Answer: Operationer var svære at forstå.

Question 4: Var der nogle besværligheder ved at omdanne illustrationerne til kode?
Answer: Nej, når man fik det mest basale på plads, gav det fint nok mening

F. Interview Transcription

Interviewer 1: Okay, vi vil gerne vente med at fortælle dig om hvad vores projekt egentlig handler om. Det handler om maskinlæring, i forhold til vores projekt. Det er også derfor at vi gerne vil snakke med dig, det er fordi du er underviser i kursus på 5. Semester. Vi vil ikke forklare præcis hvad vi gerne vil, vi vil gerne undgå at påvirke hvad du siger til os. Men vi fortæller løbende mere om vores projekt, og starter med at stille nogle spørgsmål angående kurset, hvad målene er, og hvad du synes er vigtigt. Derefter lige så stille gå ind mere og mere i hvad vores projekt egentlig handler om, og stille løbende spørgsmål til det.
Den interviewede: Ja, jeg kunne tænke mig at vide er det her et spørgsmål om at interviewe forskellige kursusholdere for at få noget generelt, information omkring pædagogiske ting, eller er det her mere at jeg skal komme ind på jeres projekt.
Interviewer 1: Det er den form at ... vores projekt i korte træk er at lave noget som til folk der underviser og folk der bliver lært omkring maskinlæring. Det er det vores projekt egentlig handler om, og dig som underviser er (det) relevant at vide hvad du synes er vigtigt, og hvad du ikke synes er vigtigt for de studerende, for at forstå hvad de ofte misforstår. Så hvad du synes som kursusholder er (vigtigt for) maskinlæring og de studerende. Derfor er du i vores brugergruppe.
Den interviewede: Så i er på 4 semester? Så i har ikke haft noget om maskinlæring?
Interviewer 2 & **Interviewer 1**: (Nej, kun I projekter).
Interviewer 1: Var det svar på dine spørgsmål?
Den interviewede: Måske lige et ekstra spørgsmål, jeg går ud fra at det her er et projekt omkring sprog også?
Interviewer 2 & **Interviewer 1**: Ja (Ja) Ja
Den interviewede: Jeg ser ikke helt sammenhængen her til det tema.
Interviewer 1: Det er jo det der er spændende. Det er jo sprog og oversættere som sagt som kurserne på 4 semester går ud på. Vi vil gerne vente med at sige præcis hvad vi laver, for vi vil gerne spørge dig løbende om du har nogle idéer til hvordan tingene skulle se ud og sådan noget.
Interviewer 1: Hvor mange år har du været på universitet

Den interviewede: 17 år.
Interviewer 1: Hvor Længe har du haft kursus i maskinlæring?
Den interviewede: Det kører lidt på skift. Vi rotationsordning hvor vi normalt siger at man har det i 3 år, så skifter man og får noget nyt ind over. På grund af bemandings mæssige årsager og ændringer i studieordningen, så har jeg haft den én eller to gange, mere end de 3 perioder man normalt er tiltænkt. Men målet er at have rotation hvert tredje år.
Interviewer 1: Så du har været det I 4-5 år?
Den interviewede: Ja.
Interviewer 1: De første spørgsmål omhandler meget om kurset og hvad der egentlig er deri. Vi kiggede på studieordningen, den virkede meget svagt beskrivende af hvad målene egentlig er med kurset. Hvad syntes er de generelle læringsmål for kurset?
Den interviewede: Jeg har også en moodle side jeg ved ikke om i har været inde og kigge på den.
Interviewer 1: Nej, det tror jeg ikke.
Den interviewede: Okay, der står inde på den side mere omkring hvordan kurset det er, og hvad der er tiltænkt.
Interviewer 1: Okay, kan være du lige kan svare kort på: Er det meget algoritme baseret, matematik baseret, eller sådan model baseret.
Den interviewede: Lige I det her kursus, er formålet - Det er jo første gang de studerende arbejder med maskinlæring hvis det ikke er direkte tilvalg igennem projekterne. Så kurserne etablerer fundamentet for kunstig intelligens. Ikke nødvendigvis maskinlæring, men hvor maskinlæring er en komponent. Så vi går ikke så meget i detaljer med de forskellige metoder, men kurset giver et kendskab til forskellige områder, helt ned fra søgealgoritmer i kontekst af intelligente agenter, til maskinindlæring systemer hvor vi kigger på både prediction, klassifikation, clustering, over til grafiske probabilistiske-modeller [05:28] til mere normative systemer hvor at der skal gives støtte til at foretage beslutninger. For eksempel,via ???-diagram [05:40]. Så det er et kursus der i højere grad fokuserer på bredden frem for detaljerne omkring de enkelte metoder.
Interviewer 1: Det giver også god mening. Når du snakker om bredden, går jeg ud fra at man bliver introduceret til en masse forskellige ting indenfor maskinintelligens? Når man så møder en algoritme eller model, når du siger man ikke går så meget i detaljerne, er det så matematikken, og den direkte implementation du henviser til der?
Den interviewede: Nej det er mere at normalt har du et område illustrere området ved at tage nogle metoder ud. De metoder er typisk blevet forfinet over årene. Hvor der kom flere ting på og lavet flere analyser af deres adfærd og lignende. Der er det fokus nok mere på de grundlæggende teknikker der er. Vi går ikke så meget ned I opfølgningen på alle detaljer af hvordan man rent faktisk bruger de her systemer. For eksempel: et

område som er en del af kurset er probabilistisk modeller. [7:06] ??????? netværk. Hvor at de at lave sandsynligheds opdateringer er et element i kurset. Til at lave de sandsynligheds opdateringer beskriver kurset nogle grundlæggende teorier for hvordan man kan gøre det, men der eksisterer meget litteratur som beskriver mere effektive måder at gøre det på. Det er ikke noget der bliver gået over i kurset. Det er så en afregning mellem at have tid til at dække de elementer fra AI kurset som er relevante uden at bruge for meget tid på de enkelte.
Interviewer 1: Er der nogle dele af kurset I springer over eller ikke gå dybt i. Det er dog måske de detaljer her eller er der andre ting du sådan tænker?
Den interviewede: Det er jo detaljerne. Det er jo et spørgsmål om hvad man ønsker. Et design parameter er jo at det bliver fulgt af mange forskellige studerende. Det bliver fulgt af software studerende, datalogi studerende, studerende der følger CS-IT, det vil sige det er studerende der kommer fra BAIT, men det kan også kan være internationale studerende som kommer fra forskellige baggrunde. Hvor der ihvertfald er to "tracks" indenfor CS-IT som følger det. Så vi har egentlig 5 studieretninger der følger dette kursus, og derfor skal man designe et kursus som henvender sig til alle. Så ideelt set, f.eks software studerende gøre mere ud af programmerings komponenten. Men det er mindre attraktivt for de andre studerende. Omvendt så er der noget omkring modellerne som ville være mere interessant for datalogi studerende end for software studerende. Der er kunsten i det her kursus at finde en passende balance af de forskellige komponenter der er. Under hensyntagen til forskellige profiler som de studerende har der går på det her kursus.
Interviewer 1: Nu nævnte du selv programmering, er det noget man gør overhovedet I kurset?
Den interviewede: Ja, og nej. Jeg har prøvet at designe - i forelæsningerne gør vi det ikke. JEg har tilrettelagt opgaverne sådan at der er nogle opgaver hvor man kan bruge software værktøjerne. Der foreslår jeg 2 forskellige man kan bruge: En der er baseret på en grafisk brugergrænseflade, og en som er en programmerings omgivelse. Sådan at man kan egentlig kode øvelser hvis det er det man har interesse i ...
Den interviewede: Sådan at man kan egentlig kode øvelserne hvis det er det man har interesse i og ellers har kendskab til. Men det fordrer et initiativ fra de enkelte studerende som ønsker at gå den vej, men mulighederne for at kunne tilgå programmering vil så komme via øvelserne så er det sådan noget som man kan vælge individuelt.
Interviewer 1: Hvilke dele af af kurset føler du som ofte bliver misforstået af de studerende?
Den interviewede: Der hvor de største udfordringer er, det er når vi snakker de mere formelle modeller, og som ligger lidt længere fra metoder

som man kunne have stiftet bekendsskab med tidligere. Og et eksempel her kunne være bayesianske netværk som i høj grad baserer sig på sandsynligheds-calculus. Det bliver introduceret i kurset i en hvis grad, men det er sevlfølgelig altid en god ide af have baggrund, det er jo også noget som indgår i programmeringsniveau. Og egentlig også noget folkeskolen de bruger og resonerer omkring sandsynligheds-calculus er typisk et af de områder der er lidt vanskligere tilgængeligt.
Interviewer 1: Nu nævner du selv at i brugte det der grafiske programmeringsværktøj, og det andet miljø som du snakker om at man kan lave opgaver i, er der andre værktøjer, programmer der anvendes i undervisningen?
Den interviewede: Altså fra de studerendes side eller fra min side?
Interviewer 1: Begge.
Den interviewede: Jamen jeg har brugt quizsystem til at få stillet spørgsmål, og så kan man via mobiltelefon melde ind hvad man tror svaret det skal være, så har jeg udviklet (men det er måske ikke så meget software) nogle screencasts som supplerende til de områder som erfaringsmæssig viser sig at være lidt vanskligere. Og så er der to typer software, der er grafiskbaserede, det er: (weka og hugin?), og det andet programmeringsomgivelsen der er baseret på Python, det er sci-kit learn.
Interviewer 1: Nu må jeg hellere spørge selvom jeg nok kender svaret: Sådan noge som maskinlæringsbiblioteker og programmeringssprog hvad bruger du der, hvis du bruger noget til de studerende og eller dig selv?
Den interviewede: Spørger du om hvad jeg bruger i mit arbejde og min forskning eller hvad jeg bruger i min undervisning.
Interviewer 1: Kurset hovedsageligt.
Den interviewede: I kurset bruger jeg kun scikit learn og weka har jeg så, som er en javaimplementation og der egentlig også er en java api bagved hvis man ønsker og kan lide at bruge java kan man også tilgå det den vej. Men det er jo så enten java eller python-baseret på enten weka api eller sci-kit learn pakken. Det er sådan de to.
Interviewer 1: Det var en af de spørgsmål som jeg havde omkring kurset. Og som du ved er vi jo på 4. semester så vi skal lave sprog og oversætter til programmeringssprog. Det som vi i meget meget korte træk, før vi forklarer den fulde ide og også gerne vil vise dig nogle eksempler på det vi har lavet. Det er at vi vil lave et programmeringssprog til udvikling af neurale netværk. Og det er hovedsageligt baseret til læring.
Den interviewede: Til læring af neurale netværk eller til læring for studerende?
Interviewer 1: For studerende, det er det jeg mener med at man måske kunne bruge det til undervisning eller folk der er entusiaster eller studerende som gerne vil introduceres nemmere til nogle af de her algoritmer, og som måske synes at nogle af tingene er lidt sværere. Vi har så valgt at

fokusere på læring som sagt, med de studerende og undervisning og så videre, i stedet for researchers og engineers fordi de ville måske ofte gå til de mere velkendte og gennemtestede værktøjer som tensorflow og diverse store kendte biblioteker. Nu har jeg selvfølgelig ikke sagt så meget om det, men hvad er din første tanke omkring sådan et projekt?
Den interviewede: Min første tanke var at jeg synes det lyder interessant at tage det ud fra et studenterperspektiv, det kunne jeg godt forestille mig, det kunne godt forestille mig kunne være noget der kunne biddrage potentielt med noget nyt. Og når jeg siger potentielt så er det fordi der jo er en del omgivelser allerede, det er jo nogle af pakkerder f.eks. er, nu koder jeg jo primært python, så det er reference om her, at de udvidelser der er til python, gør jo egentlig et probobalistisk sprog til et probobalistisk programmeringssprog. Og der er jo nogle af de her pakker som netop er designet til at være lette at anvende. Og et gængs eksempel er jo f.eks. keras som jo godt nok baserer sig på tensorflow, men hvor man jo kan konstruere et neuralt netværk lag for lag, ved at have en linje kode for hvert lag, hvor man simpelthen bygger modellen naturligt op. Nu ved jeg ikke, er i bekendte med den omgivelse?
Interviewer 1: Jeg har ikke brugt Keras nej, men vi har kigget på man kodeeksempler i TensorFlow hvordan de kan lave det lag for lag.
Den interviewede: Grunden til at jeg siger Keras og ikke TensorFlow er fordi keras er en abstraktion over tensorflow der gør at det er meget nemmere tilgængeligt, og nu siger du at det er fordi i kigger i retning af studerende. Selve specifikationen af et neuralt netværk så kører det på en enkelt linje hvor man så siger den første er et inputlag, og så angiver man hvor mange neuroner der skal være i det lag, okay næste linje det er så næste lag, hvor mange neuroner skal der så være der og hvilken aktiveringsfunktion skal den anvende, så man bygger det op så hvis i har en kompleks model med 10 lag f.eks. så kan det repræsenteres med 10 linjers kode. Så en udfordring, nu går det så tilbage til det der potentiel, det vil så være hvordan i har tænkt jer at positionere jer i forhold til de eksisterende programmeringssprog til uvikling og specifikation af neurale netværk. Så det vil nok være min ummidelbare holdning.
Interviewer 1: Vi har lige nogle få spørgsmål før, så vil jeg gerne svare på dit spørgsmål. Okay, hvis nu du skulle tænke på at du skulle lave et programmeringssprog til udvikling af neurale netværk i sådan et studieorienteret miljø, hvis det var en perfekt verden og du kunne få alle de features du ville have, hvilke slags features skulle der så være i sådan et sprog, og måske hvad skulle abstraheres væk og hvad skulle ikke abstraheres væk og hvad skulle ikke abstraheres væk? Et eksempel kunne være matematik-delen, specifikke implementation, noget med feed-forward og backpropagation algoritmerne, og noget dataflow eller et eller andet.
Den interviewede: Nu skal det lige siges at det her er udfra at jeg får

præsenteret spørgsmålet uden at have tid til at tænke over det, men for det første så vil jeg sige at man ikke skal gå efter alle mulige features, så man skal ikke gå efter så mange features som muligt, fordi det tror jeg generelt at det ville forvirre mere end det ville gavne, så det er et spørgsmål om at zoome ind på de features der rent faktisk er målet for denne programmeringsomgivelse, og det vil så sige formålet for de studerendes læring, og det er så igen med hvilket niveau de studerende er på. Hvilken baggrund har de fra tidligere. Hvis vi antager at det er 5. semester, det henvender sig til studerende der har det her som deres første kursus (I maskinlæring) og de bliver præsenteret for denne omgivelse, så ville jeg nok fra programmeringsdelens side sige at niveauet med keras i forhold til konstrueringen modeller er god fordi den i bund og grund er en 1:1 afbildning mellem kode og så strukturen i netværket.
Den interviewede: Niveauet I keras I forhold til konstruering af modeler er god fordi at den ??? en til en afbildning mellem kode og så, strukturen I netværket. Der hvor man kunne . . . forestille sig nogle . . . udvidelser det ville være at gøre, suplere kode med en grafisk komponent, sådan at man kan se hvordan, koden afspejler sig I en eller anden grafisk struktur. Det . . . burde være meget nemt at gøre. Den næste del ville være at støtte mere op omkring, illustrere hvordan indlæring den foregår, med hvad er det for nogen gradienter der bliver beregnet, hvis det er backpropagation. Og det vil jo så være en illustration I ??? kædereglen for differentiation. Men hvordan de her komponenter de bliver sendt op igennem systemet som, sender nogle beskeder ned, og op igen.
Interviewer 1: Ja, netop.
Den interviewede: Få det visualiseret, kunne jeg forestille mig det kunne være. . .
Interviewer 1: Når du siger "få det visualiseret", tænker du så syntaktisk eller grafisk.
Den interviewede: Det vil nok mere være. . . Det vil nok mere være grafisk. For programmeringssprog, altså det du ser det er, de gemmer jo alt det her omkring backpropagation og så videre, det bliver jo gemt væk. Så det er ikke noget man sidder og koder. Det kan du gøre, men, typisk er det en del af omgivelsen. Og jeg. . . når jeg tænker I retning af dette her, så tænker jeg heller ikke at studerende skal side og kode, backpropageringsalgoritmer. Jeg tænker I retning af at, I hvert fald for et kursus som mit, at det vi være mere at illustrere hvordan, backpropagering den foregår.
Interviewer 1: Ja
Den interviewede: I et netværk de selv har konstrueret.
Interviewer 1: Ja. . . . så det er vigtigere at vide hvordan det her signal bevæger sig igennem sit netværk end det er hvordan den reelt set gør det. Er det du prøver at sige eller mener du...

Den interviewede: For mit kursus, fordi at det vil jo simpelt hen, det er uden for, det vil ikke være, vi vil ikke kunne nå at gennemgå så meget.
Interviewer 1: Det er en for stor detalje, eller hvad.
Den interviewede: Ja, I hvert fald hvis det skal gøres... ja, på nuværende tidspunkt vil jeg sige det ville være for meget, at gennemgå. Det er heller ikke del af pensum, for eksempel, at have en fuld forståelse for elementerne I backpropagation. Det er mere, formålet er mere, en del af pensum er at fange, I hvert fald intuitionen bag backpropagation og, sådan, de generelle principper der ligger til grund for det, uden at kunne gennemgå et... eksempel.
Interviewer 1: Ja. Okay, nu er vi egentlig nået til hvor vi gerne vil vise dig vores sprog. Og lige komme med en hurtig introduktion til hvad det er vi gerne vil. Og det er heldigvist langt hen ad vejen nogen af de ting du nævner faktisk. Vi vil gerne lave et dataflow orienteret programmeringssprog. Ved du hvad det er?
Den interviewede: Neeejjjj... ikke lige...
Interviewer 1: Sådan lige I korte træk... Tensorflow er faktisk datafloworienteret, hvis du ved dens implementation. Men det der sker er man... presenterer sit program, som en... hvad heder det? “directed graph”, hvad heder det på dansk?
Den interviewede: En orienteret graf.
Interviewer 1: Hvad siger du?
Den interviewede: En orienteret graf.
Interviewer 1: Ja, så man presenterer sit program som en orienteret graf.
Den interviewede: Ja?
Interviewer 1: Hvor... du så har noder... vi kalder dem så blokke, hvor noderne så er en eller anden form for, computation, altså beregningsenhed. Og... kanterne så er forbindelser mellem de her noder. Og berengelser sker I noderne, og så overfører man så signalet fra node til node.
Den interviewede: Mhm.
Interviewer 1: Og en node den fyrer så lige snart alle dens... alle forbindelser der går ind I den er klar, så fyrer den så videre. Så det vi gerne vil gøre er at lave et programmeringssprog, som, er datafloworienteret, fordi mange af de modeler man ser, hvis du tager sådan noget som et LSTM netværk hvor man ser de modeler, de har faktisk de her... blokke af ting og nogle forbindelser imellem dem, så det er ligesom tegnet dataflow orienteret. Så vi laver et progammeringssprog så... gør afstanden imellem implementation og model mindre, så du kan se modelen, og skal du implementere den som du ser den. Giver det, mening?
Den interviewede: ...Ja... altså en beregningsgraf er jo også, det er jo sådan en illustration af de beregninger der foregår I et neuralt netværk, så der er sådan mere eller mindre en til en afbildning.
Interviewer 2: [Viser illustration] Det er de her slags modeller som vi,

som har baseret det på.
Den interviewede: Ja.
Interviewer 1: Ja, der har vi, et billede af LSTM laget her, hvor vi kan se der forskellige noder, og nogle kanter imellem, som viser nogle... retninger og nogle forbindelser. Og beregnelserne sker i noderne. Og vi vil så basere et programmeringssprog på det samme premis, hvor syntaksen bedre komunikerer hvad du ser, så der er en kortere vej imellem, modellen og din kode. For ligesom at gøre det nemmer at implementere model du aldrig har set før ud fra modellen, selve den grafiske model.
Den interviewede: Ja.
Interviewer 1: Så vil vi gerne vise dig et [Afbrudt af følgende] kodeeksempel...
Den interviewede: Jeg skal lige, altså, bare lige for at forstå niveauet I snakker om. Så i vil gerne have sådan at man kan... Ok... for eksempel I Keras, så vil man have en... en kommando som egentlig bare heder LSTM, hvor at den so klarer alt det her for jer. Men I ønsker så at kunne, på en eller anden måde få altså at kunne, man kunne lave sin egen LSTM celle, og måske lave ændringer i.
Interviewer 1: Tanken er at vi også gerne vil abstrahere meget af matematikken og backpropagation, og feedforward væk egentlig, alle de der, algoritmiske og matematiske komponenter, det vil vi gerne abstraheres meget væk, du skal ikke fortælle, hvordan, fx matrixmultiplikation fungerer, du skal bare sige at her skal ske en matrixmultiplikation. Så har du en, måske en matrixmultiplikationsnode, som er... supleret af sproget. Og så siger du så vi skal så have vores input og vores vægte, ind I den node her, og får det så ud, og så ind i, noget bias og så Sigmoid for eksempel hvis det er et helt normalt lag. Den måde at repressentere et neuralt netværk i form af den her graf her, og så kan du så opbygge dine egne noder, og sætte noder sammen, på den måde så kan du definere hele dataflowet i den her model, uden at implementere matematikken. Og det bliver ligesom taget hånd for dig, det der med backpropagation og feedforward og det, du skal bare definere dataflowet, og hvad der skal ske, men ikke hvordan det skal ske... Ja... Giver det ok mening? **Den interviewede**: Ja, så vi har en graf. Bare lige for termonologi her. Jeg tror man kalder det knuder på dansk.
Interviewer 1 [oven I hinanden]: Ja, det tror jeg også man gør, det tror jeg du har ret i.
Den interviewede: ...[overdøvet]...en node er noget med musik og sådan noget.
Interviewer 1: Ja, det har du ret i.
Den interviewede: Så de knuder der er I grafen de repressenterer så resultat af en operation, og så kan man lægge operationer ind fx matrixmultiplikation og man kører den igennem en Sigmoid funtion, og det knytter

de forskellige knuder sammen.
Interviewer 1: Ja, netop. Så du har ligesom de forskellige knuder, og nogle knuder som er det vi kalder, vi har to typer knuder I vores sprog, vores sådan syntaks: Vi har det der hedder blokke og det der hedder operationer. Operationer er der hvor selve matematikken den sker. Og blokke er ligesom en wrapper for operationer og forbindelser derimellem.
Den interviewede: Ja.
Interviewer 1: Så du bygger blokkene, og vi suplerer operationerne. Og så kan du bygge dine egne knuder, altså blokke, og så forbinde dem som du vil og genandvende dem.
[Ser på diagram?] **Den interviewede**: Ja.
Interviewer 2: Ja.
Interviewer 1: Ja.
Interviewer 1: Netop.
Interviewer 2: Ja
Interviewer 2: Det at man kommer... have blokke inden I blokke, så man... laver én LSTM blok, og så kan man I en ydre blok komme... nogle LSTM blokke sammen på en måde der passer i et netværk.
Interviewer 1: Lige åbne et program så du kan se, hvordan vi ligesom tænker det kan se ud, sådan noget syntaks... [...]
Interviewer 1: Kan du nogenlunde se?
Den interviewede: Jaa.
Interviewer 1: Så det der sker det er vi har nogle forskellige komponenter I vores sprog her... Blok er jo I virkeligheden en glorificeret klasse hvis man kender OOP. Så vi definierer en blok som vi kalder "ANN layer", hvis har det der heder gates, som ligesom er der vi kan lave vores forbindelser til og fra. Vi har selvfølgelig inputgates og outputgates. Så har vi det der heder et buleprint, hvor du ligesom definerer dataflowet I den blok her. Og der har vi nogle sources... en source er en knude uden input, kun et output. Som ligesom suplerer en matrix eller et signal. Et signal er en matrix I vores sprog.
Interviewer 2: Som et bias for eksempel.
Interviewer 1: Ja, netop. Så vil vi definere vægte og bias, og så kan du så lave en multiplikationsknude. Og så kan du så forbinde dit input, og vægtene til multiplikation her. Lav en additionsnode og så at det er biaset ovenpå, og så køre igennem et Sigmoid lag, forbinde det med de forbindelser her. Så de her pile repressenterer ligesom de her forbindelser, I de her grafer her. Og på den måde så kan du egentlig implementere et fuldt neuralt netværk lag, på, ja hvad, 30 linjer, med mellemrum og nogle kommentarer. Og så kan du så genanvede den blok her til at lave et helt neuralt netværk, og forbinde dem I en lige linje.
Interviewer 1: Eller bare tilføje, bare lave et lag, ja. Har du også spørgsmål herovre egentlig? De ligger på discord hvis det er. Hvis du lige skulle, nu

ser du bare på det første gang selvfølgelig. Øhhm, kan du se pointen med sådan noget her som underviser? Kan du se at man måske som studerende kan få noget ud af det her?
Den interviewede: Ja, det kan man godt.
Interviewer 2: Er der noget der står ud som specielt godt eller dårligt?
Den interviewede: Altså, der er nogen ting, den positive side af det, jeg synes faktisk det er en spændende måde, i har valgt, at illustrere flowet af det med pile. Det støtter rigtig godt op med at få den her graf forståelse. Jeg synes også at det her med at i bruger de her operationer fungerer godt. Der er lige lidt afstand imellem operationer og hvad der rent faktisk sker, men det er i småtingsafdelingen. Hvis nu vi tænker i retning af at det skulle være for en studerende. Der er en eller anden, en ting der ikke er helt klar, det er input / output på den her block blueprint. Der kommer noget input ind udefra, fra source af.
Interviewer 1: Nårh, ja, okay. Hele blokken er den her block, som er outer scope for blueprintet, som definerer hvordan en block opbygges.
Den interviewede: Okay, men hvad er så input og output, men hvad er input, er det så, input det er så source og weight?
Interviewer 1: Et input det er et signal fra en anden knude, så det er en, vi har en eller anden kant fra en knude til den her knude, som ligesom går ind i dens input-gate. Så vi ligesom supplerer et signal, så det er ligesom svaret at en forrig knude i den her graf her. Så det er en matrix, i det her tilfælde. Så den vil få en matrix som input, også kan den så lave nogle operationer og nogle beregninger på det her input, og så kan man så forbinde det svar til output og så vil alle knuder der senere hen i den her graf er forbundet til den her knude, vil så få at vide at nu er den her knude klar, og så kan man bruge dens output, som dens eget input og ligesom lave de her forbindelser.
Den interviewede: Ja, okay. Så, men jeg vil måske lige sige, det er ikke klart her, hvad der er input og hvad der er output.
Interviewer 1: Ja.
Den interviewede: Hvis jeg lige kigger på det her, er der nogle krav til det, og det er måske i en lidt.. Lidt mere generelt man kunne sige, det er at, altså, for eksempel, hvad er dimensionerne på det her? Det har i valgt at skjule her. Det kan have sine fordele fordi for nogen, hvis man siger, det er ikke nødvendig viden her, men det skjuler også lidt hvad der så rent faktisk sker.
Interviewer 1: Ja.
Den interviewede: Og det ved jeg ikke. Det kunne man godt overveje i hvert fald, om det er noget som er hensigtsmæssigt for studerende. Det afhænger af hvad man skal lære.
Interviewer 1: Vi har faktisk også i sproget, ikke i det eksempel her, der har vi en type, der hedder size, som er to ints, rækker og kolonner i en

matrix, så man kan sætte størrelsen af de her signaler. I det her eksempel er de dog ikke en del af dem, men vi har snakket om at man skulle kunne sætte dem, eller man skal kunne lade være.
Den interviewede: Yeah.
Interviewer 1: Så det er en del af sproget. Men det kan være du synes det er noget man bør overveje at have fast.
Den interviewede: Det er sådan lidt et spørgsmål om, at snakke om typer. Det er jo i bund og grund det samme her. Jeg tror, at for at forstå, hvad der sker, når man snakker om at skulle lære sit første programmeringssprog. Så tror jeg det er en god idé at arbejde med de her typer. Det gør også at man kan debug.
Interviewer 1: Helt klart.
Den interviewede: Men altså, i forhold til. Jeg synes det afhænger af hvad det er, hvem man henvender sig til. Hvad er målet med det? Hvad er det de studerende skal lære hvis de bruger de her? Hvis man sammenligner, nu bruger vi bare Keras, som eksempel igen, fordi at den kan gøre det her ret kompakt, så det er igen med at, der kan man egentlig lave sit eget lag, med én linje af gangen, hvor man også kan angive størrelsen på de linjer, så altså hvor mange skjulte ord der skal være. Og, som, i hvert fald med hensyn til, det viser godt synes jeg, hvad er det for nogen, hvordan et neuralt netværk det er bygget op. På den anden side, hvis du for eksempel lægger et LSTM lag ind, så skjuler den, hvad det er for nogle operationer der foregår der. Men der sådan noget, som jeres sprog formodentlig ville kunne illustrere meget nydeligere. Hvad det egentlig er der foregår. Og det vil så selvfølgelig kunne have fordele, hvis man skal bruge det her til at illustrere for de studerende, hvad det er for nogle operationer der skal foregå, men så på den anden side, så er der også det her med vi udelader typerne, eller de dimensioner, på de matricer, som der er.
Interviewer 1: Håbet er, så også at når man så har defineret en block som sådan et lag her, så kan du jo i en anden block, som så er et netværk, der kan du også lave et lag på én linje. Ved bare at, det vi kalder build, den block her. Så når du har defineret laget, så kan du faktisk også lave det på én linje i andre blokke, så lave det som en anden block, som anvender de blokke her. Så det er jo ligesom, håbet. Eller tanken i hvert fald hedder det rettere sagt.
Den interviewede: Så kan i også lave automatisk type detektion ud fra det her, så længe at vi ved hvad vægtene de er, altså når i kender input og i kender vægtmatricerne, så kan i så også udlede, hvad output er. Dens type og dimension.
Interviewer 1: Ja. Netop. Men, man kan godt mærke på dig, du måske synes, at det er lidt en skam, at man gemmer størrelserne væk, eller er det forkert forstået af dig?
Den interviewede: Nej, det vil jeg ikke sige, jeg er simpelthen lidt i...

Det er jeg uklar på.
Interviewer 1: Sådan lidt splittet?
Den interviewede: Ja, det er et spørgsmål om hvem igen, man helt præcist ønsker at fange. Jeg synes der skal være mulighed for at kunne finde størrelserne, eller finde.. Er det noget der skal kunne udledes i systemet, eller er det noget, som brugeren skal, at det kræves af brugeren at man angiver typerne undervejs. Hvad der fungerer bedst i en indlæringssammenhæng, det ved jeg faktisk ikke. Altså, det ville jo være nemt, at parse det her og så bagefter finde ud af, hvad er det for nogle typer vi skal ha'.
Interviewer 1: Ja, netop.
Den interviewede: Men omvendt, at tvinge brugeren til selv at angive typer, gør jo også, at man begynder at tænke mere over, hvad er det egentlig for nogen operationer der skal være med. Så hvad der er mest hensigtmæssigt af det, det ved jeg faktisk ikke, altså det der med, at hvis man bare trykker, så får man at vide, hvad er typerne, nårh ja, det vidste jeg jo godt. Det var sådan, da det var, at okay.
Interviewer 1: Så du kan se fordelene i begge retninger eller hvad? Eller ulemperne eller?
Den interviewede: Ja, det er jo det jeg siger, det afhænger lidt af, hvad kan man sige, man ønsker med det her.
Interviewer 1: Okay, nu har vi jo valgt, at, ligesom, at Keras også gør, at gemme nogle, masse ting væk fra den studerende. Især alt matematikken har vi sådan taget væk. Og vi har også feedforward og backpropagation er også gemt væk. Og det bliver jo indirekte defineret af de her forbindelser i virkeligheden af de her kanter. Når man har defineret dataflowet i den ene retning, så har man også det omvendte i den anden retning.
Interviewer 1: Når man har defineret dataflowet i en retning, så har man også det omvendte i den anden retning.
Interviewer 1: Synes du vi har gemt for meget væk, eller er der noget du synes vi har gemt for lidt væk?
Den interviewede: Narh, jeg ved ikke om jeg synes i har gemt så meget vækj igen, f.eks. i jeres multiplikation.
Den interviewede: Der skal man have kenskab til matrix multiplikation for at man ret faktisk skal kunne skrive det der op. For hvis input dimensionerne ikke passer, så får man heller ikke den rigtige output dimension.
Den interviewede: Så... Det kan godt være at det ikke lige er i sproget, men det kræver stadigvæk en forståelse for... Nej jeg synes ikke i har... DEt det synes jeg ikke i har.
Interviewer 1: Så omvendt, har vi gemt for lidt væk? Noget du tænker vi burde have gemt væk?
Den interviewede: Nej, det ikke kan jeg ikke umiddelbart komme på, men lige for at få det der med hvor i har, det man måske har gemt lidt for lidt væk af / for meget væk af.

Den interviewede: Det er det her igen med typerne, især sådan noget med vægtende heroppe, "Build source build source"...
Den interviewede: Det afhænger igen af hvad i vil, der skal lære det her, lærer man så virkelig hvad der sker hvis man bare har en kommando der hedder "Build source"
Den interviewede: Uden at man skal forholde sig til hvad der sker her?
Interviewer 1: Altså inden i Sourcen?
Den interviewede: Ja, erhm.....
Den interviewede: Og altså der bliver jo et eller andet, hvis nu i f.eks. har to matrix multiplikationer efter hinanden, så skal vi jo have dimensionerne på de matrix multiplikationer til at stemme overens, ellers så fungerer det ikke.
Den interviewede: Nu sagde du at det kunne, at man kunne angive noget med typer heri. Det vil der jo på en eller anden måde være behov for at kunne udtrykke sig for, på en eller anden måde...
Den interviewede: Programmet på den måde er... Som det står her nu, så ville jeg.. Jeg tror ikke at, så kan man jo ikke...
Den interviewede: Angive hvad... Angive 2 på hinandenen efterfulgte matricx multiplikationer.
Interviewer 1: Ikke i compile time, men i runtime kan man godt få det til at køre.
Interviewer 1: Hvis man tager et sprog som java og du har 2 klasser, MAtrix objekter, og du har en multiplikations funktion på dem, den tjekker først i runtime om de passer.
Interviewer 2: Ja men også.. Der er vist erhm... Vist man bare har det input man kommer til at få, hvis man har størrelsen på det, så er der rigtig meget af størrelserne igennem systemet som man igennem systemet kan komme til at finde ud af i compile time.
Den interviewede: Ja men det afhænger vel af hvad de vægte er her.
Interviewer 1: Ja det er rigtigt, du har helt ret i source. Det er nok meningen at der bør være en size i virkeligheden.
Interviewer 1: Ved source bør der nok altid være en størrelse.
Den interviewede: Ellers så er det i hvertfald underspecificeret
Interviewer 1: Ja det tror jeg du har ret i. Det rigtigt observeret. Vi har også således at i blueprintet kan du give parametere med, ikke i dette eksempel, men man kan give en size parameter med, håbet er så at man kan anvende dem til at dynamisk lave størelserne igennem dem.
Interviewer 1: Sources de bør være specificeret.
Den interviewede: JEg er ikke sikker på at det er nok. ikke hvis i har kompleksre struktureer, hvis i har... Hvis jeg kender input og jeg kender output, og jeg kun har en matrix multiplikation, så det nemt hvad jeres matrice skal være, så bliver det jo bare et spørgsmål om at have input og output til at stemme overens.

Den interviewede: Hvis man har 2 efterfølgende matrix multiplikationer så svarer det jo til at transformere det ned til et rum, og så over til et andet rum igen. Hvor det første... Det midterste rum er jo ikke angivet, hverken af input eller output.
Den interviewede: Så der skal noget mere til for at sige det.
Interviewer 1: Ja okay.
Den interviewede: Altså hvis i har X gange A gange B, hvor x er input, a og b er to matricer, så ved jeg at hvordan min output og input dimensioner er der. Men jeg ved ikke hvordan den transformation der foregår efter jeg ganger...
Den interviewede: Jeg ved... Okay sagt på en anden måde, hvad er kolonnerummet af a matrixen? Den er kun betinget af rækkerummet på B matrixen.
Den interviewede: Men det har jo intet med input og output at gøre.
Den interviewede: Jeg tror altså i skal ...
Interviewer 1: Hvis man havde defineret A gange X gange B så ville man jo kende det ikke sandt? Er det ikke hvad du prøver at sige? At man bør kende dem?
Den interviewede: Jo jeg siger bare at i skal kende, hvis det her skal virke generelt, og i skal have fuldstændig frihed inde i disse blueprints, så skal vide andet end dimensionerne på input og output, så skal der være noget om disse source knuder.
Interviewer 1: Ja netop,... Ja.
Interviewer 2: Måske ikke nødvendigvis på hver blok at man skal sætte nogle faste størrelser, men istedet alle steder hvor der kommer noget data.
Interviewer 2: Der hvor der kommer tal ind i systemet, der skal man have styr på hvilken størrelseorden de her tal/matricer har.
Den interviewede: Nogen gange kan man udelede det. I dette tilfælde har man ikke brug for weight og bias, det kan man udelede ud fra de andre, eller ihvertfald dimensionerne.
Interviewer 1: Men hvis det er en kæde af dem, så bliver det problematisk. Ja....
Den interviewede: Så umiddelbart ville jeg nok sige, det er nok det...
Interviewer 1: en detalje man bør overveje?
Den interviewede: Ja nu var spørgsmålet om vi pakker for meget væk. Og lige i dette tilfælde er der i hvertfaldn noget der er et behov for at specifivere.
Den interviewede: Og så er der noget om hvor meget man skal have med til læring, men det er i hvertfald noget man skal have med som minimum.
Interviewer 1: Vi har måske allerede, vi har måske allerede været ved det her spørgsmålet igennem de andre ehh...
Interviewer 1: Men noget du tænker bør ændres? Noget som ikke har noget med at pakke for meget eller for lidt? En syntaktisk ting, som du

måske tænker bør overvejes at ændre?
Den interviewede: Altså... Hmmm.... Det måske lige om det her om blueprint ikke tager nogle blueprint parametre
Interviewer 1: Det kan det godt
Den interviewede: Men det kan det så gøre... Erhm... Altså.... Normalt ville man jo nok have angivet i sit.... I sin funktions definition ville det jo lægge lidt op af... Hvilke parametere den skal kunne tage.
Interviewer 1: Hvad tænker du på præcist?
Den interviewede: Altså hvis jeg erhhh... Sagt på en anden måde, hvor i denne kode kan jeg se at blueprint faktisk kan tage imod nogle parametre? Altså formelle parametre?
Interviewer 1: Ehhh.... Jeg er ikke sikker på at jeg forstå... Undskyld.
Den interviewede: Når du siger blueprint godt kan tage imod parametre, formelle parametre hvor kan jeg se det henne?
Interviewer 1: I dette kode eksempel har vi valgt ikke at gøre det, det ville være meget naturligt som meget andet, så skriver man bare her oppe en type og variable navnet og så komma, som man også gør i mange andre sprog som har typer.
Interviewer 1: Så definiere du bare parametrene heroppe lidt lige som en funktion eller en konstruktor som vi kender det fra programmering.
Den interviewede: Ja men hvilke parametere, hvor mange skal den kunne tage?
Interviewer 1: Det definere programmøren jo.
Den interviewede: Okay, så der er ikke nogen restriktioner på sproget?
Interviewer 1: Nej det bestemmer programmøren helt selv, afhængigt af hvilke typer blocks de har lyst til at lave.
Den interviewede: Okay...
Interviewer 1: Ja så det er helt op til programmøren og hvordan de vil design det.
Den interviewede: Og så ehh... Så retur værdien fra blueprint funktionen... Er så egenligt defineret ud fra nogen der kaldte variable, som vi har heroppe?
Interviewer 1: Det er alt sammen procedures, i virkeligheden, der er ikke nogle funktioner i vores sprog. Der er ikke en funktion der tager et input og returnerer et output, der er kun procedures, der kan tage inputs, og så gøre noget. I den forstand at du har en blok, og når du er inde i blueprintet, så kigger vi inde i blokken, og vi kan ikke se ud. Vi kan kun se at vi har nogle gates der modtager nogle signaler, og vi har nogle gates hvor vi kan sende nogle signaler ud. Og så kan vi kigge uden for blokken, hvor vi kan anvende blokken, og connecte ind i blokken, og tage dens output og connecte til noget andet, men vi kan ikke se, hvad der er inde i blokken. Så det er altså abstraheret væk, så disse gates er altså kommunikationsvejen mellem det interne og det eksterne. Så et blueprint

definerer, hvordan en blok skal opbygges, og hvordan de her forbindelser går mellem de interne blokke, denne blok har. Og man kan have flere inputs og flere outputs. Så det er hele er defineret ud fra disse gates, hvor den får et signal ind, behandler det, og så giver et signal ud.
Den interviewede: Igen en terminologi, en gate, i hvert fald hvis man tænker på LSTM'er har jo en lidt anden fortolkning end I har her.
Interviewer 1: Ja, der har man 4 gates i den normale model.
Den interviewede: Ja, så det er en operation i sig selv.
Interviewer 1: Ja.
Den interviewede: Og det er jo lidt noget andet her, fordi det er jo egentlig mere en kanal, at der en input-kanal og en output-kanal.
Interviewer 1: Så kan måske kalde det en channel, eller sådan noget, er det det du foreslår?
Den interviewede: Det kunne måske... For folk der allerede er bekendt med LSTM'er, ville det nok være lettere at fortolke, fordi det ville være sådan lidt misvisende at snakke om gates.
Interviewer 1: Det kan du have helt ret i. ... Jeg tror faktisk at vi er det at... Okay lige det sidste spørgsmål i virkeligheden. Som underviser, måske ikke nødvendigvis kun på 5. semester, men måske også på de senere semestre, kan du så se idéen med et sprog opbygget som det her, og at man kan få noget ud af det, mere end det Keras giver, i form af lag-lag-lag - netværk. At man egentlig selv skal ind og definere dataflowet for de indre dele af det, også.
Den interviewede: Ja, det kunne man måske godt, altså jeg kan faktisk ikke huske, hvordan, og om Keras tilbyder at man selv kan lave sine egne blokke på den måde som I gør det her. Så på den måde kunne det måske godt være meningsfuldt. Jeg kunne forestille mig den (Keras) kan, men jeg ved ikke hvordan det ville sammenligne med jeres her.
Interviewer 1: Når du læser vores syntax her, synes du så at den er mere kommunikerende i forhold til selve modellen, vi kan jo lave specialiseret syntax til formålet, mere end et general purpose sprog som Python kan. De skal jo kunne lave alt, hvor vi kun skal kunne lave det her. Synes du at den syntax her er mere eller mindre kommunikerende for hvad man reelt set laver, konceptuelt.
Den interviewede: Jeg synes det måske er mere... Nej, jeg kan faktisk godt lide jeres pil-notation her, det synes jeg faktisk er en god idé. Jeg ville måske, hvis jeg skulle præsentere det her for studerende, så tror jeg at jeg ville organisere koden anderledes, jeg ville lade operationerne definere i begyndelsen, og så have netværksstrukturerne komme lige efter hinanden, fordi hvis man skal se på det her, så har vi ting der foregår mellem de her operationer, der definerer beregningskraften.
Interviewer 1: Bare så vi er sådan nogenlunde enige, så du tænker at man kan gøre noget lignende, simpelthen at rykke de her linjer op.

Den interviewede: ja, og så også de næste her, med addition og sigmoid funktionen.
Interviewer 1: Ja altså rykke dem her op, og definere dem samtidig.
Den interviewede: Ja
Interviewer 1: Og så have vores forbindelser hernede til sidst.
Den interviewede: Jeps.
Interviewer 1: Så man ligesom har forskellen mellem... Ja, vi skal måske gøre det bedre end det her, men så du mener at man ligesom grupperer de forskellige knuder i ens blok der, og så definerer forbindelserne bagefter.
Den interviewede: Ja, for så har man: Det her, er de typer operationer der indgår i den her blok, og det her er den måde blok strukturen ser ud på. Det er det der sker her. Ellers så bliver det sådan, lidt for blandet sammen. Og nu kan jeg bare kigge på de tre linjer her, og så ved jeg præcis hvordan strukturen er, specielt hvis jeg har valgt nogle sigende operationsnavne.
Interviewer 1: Netop, okay. - Har du nogle sidste kommentarer, for ellers så er vi ved at nå en ende af hvad vi har.
Den interviewede: Nej det har jeg nok ikke lige umiddelbart, altså man kan sige afhængigt at hvor kompliceret man ønsker at gøre det, så er der mange ting... der er jo et helt univers også i forhold til hvad for nogle force funktioner man bruger, og hvordan de her opdateringer foregår. Men det er ikke rigtigt noget vi har snakket om her, men der kunne man overveje hvordan det ligesom skulle understøttes, hele det læringselement der, men det er jo en helt anden boldgade. - Hvad kompilerer I ned til?
Interviewer 1: Vi er undervist i Java, og vi har allerede udviklet et lineær algebra bibliotek, og et maskinlæringsbibliotek til neurale netværk, på tidligere semestre, så lige nu tænker vi på at compile det ned til Java, og så anvende de biblioteker som vi allerede har der, til rådighed, vi har lavet igennem tidligere semestre. Man kunne også overveje sådan noget som Python, fordi det er meget kendt blandt folk der laver sådan noget her, men Python det kører vist langsommere generelt, end Java. I hvert fald af hvad vi læser på det store internet.
Den interviewede: Det er vel nok rigtigt, men altså hvis i for eksempel koder ned til... lad os sige at i brugte TensorFlow... så er det bagved liggende kode, er jo så ikke Python, det er så en implementation af det, der ville man jo så kunne udnytte de visualiserings egenskaber, der er stillet til rådighed, hvor I kan direkte visualisere den underliggende beregningskraft, hvordan den tager sig ud i det samlede netværk. I kan gå ned og monitorere hvordan de forskellige parametre ændrer sig, når I laver ændringer eller lignende. TensorFlow stiller et bibliotek til rådighed, der hedder TensorBoard, hvor I får en grafisk repræsentation af det hele i jeres browser. I skal bare køre den på Local Host, og i kan følge med i hvad der sker i konvergering og lignende. Det er en meget lækker omgivelse, som

bliver stillet til rådighed der.
Interviewer 1: Ja det er rigtigt, vi har været inde og kigge på det, og ja, det er rimelig sejt.
Den interviewede: Ja, det ville jo være et andet sprog at oversætte det til.
Interviewer 1: Ja, det er i hvert fald værd at overveje. Men jo, tak for hjælpen. Kender du nogen, vi måske kunne overveje at snakke med.
Den interviewede: Altså der er jo også Manfred Jäger, han sidder ved siden af mig, han har også undervist i MI tidligere, men det er ved at være nogle år siden.
Interviewer 1: Okay, vi fik nemlig dit navn igennem en sekretær, og spurgte, da vi jo skal have kurset her, hvis man vælger det, næste semester.
Den interviewede: Ja, jeg tror det er det. Men ellers så skulle I snakke med Bent Thomsen, fordi jeg ved at han tidligere har haft et projekt hvor det også handlede om udvikling af programmeringsomgivelser, og jeg tror også det handlede om neurale netværk. Han ville måske være interessant at snakke med.
Interviewer 1: Okay, spændende.

G. Statement objects

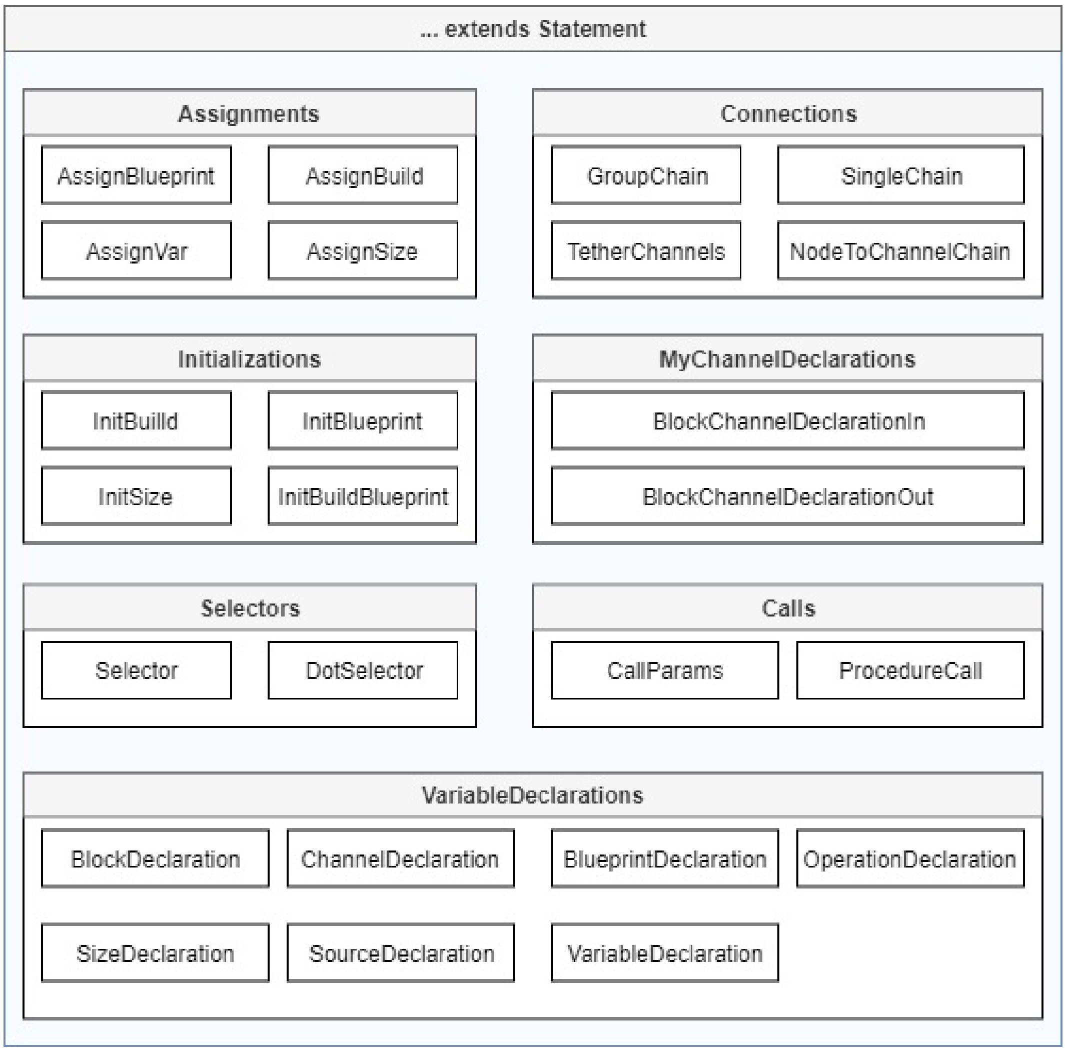


Figure G.1: The full list of statement objects.