Multi Agent System
Programming Language

Arongadongk

Alex B. Andersen, Bjarke H. Søndergaard,
Kim A. Jakobsen, Magnus S. Raichenauer,
Mikael Midtgaard & Rasmus V. Prentow.

AALBORG UNIVERSITY

The department of Computer Science
Student report
Selma Lagerlöfs Vej 300
Telephone +45 9940 9940
Fax +45 9940 9798
http://www.cs.aau.dk/

Title:
Arongadongk Programming Language

Theme:
Language Technology

Project period:
SW4, spring semester 2011

Project group:
S406A

Participants:

Alex Bondo Andersen

Bjarke Hesthaven Søndergaard

Kim Ahlstrøm Jakobsen

Magnus Stubman Reichenauer

Mikael Midtgaard

Rasmus Veiergang Prentow

Advisor:
Rong Pan

Page count: 93

Appendices count: 3

Finished: 27/5 – 2011

Synopsis:

The Arongadongk language, with its appurtenant compiler, is designed with the purpose to relieve the programmer of trivial tasks between him and his goal. The goal is to find out which local rules to apply to a set of agents in order to make them enact his desired global behavior. This is achieved by constructing a sequential imperative language with a platform independent compiler which implements a GUI and a mechanism for automation of the agents.

Preface

We would like to thank Jorge Hernandez, Hans Hüttel, Bent Thomsen, and our supervisor Rong Pan for helping and supervising throughout the project.

When reading this report you will encounter the following notations:

Headers are written at the beginning of every chapter. They briefly describe what the given chapter contains and looks like the following:

This is a header, which is at the beginning of every chapter.

Quotations are the words of another person followed by a citation to the source. This looks like the following:

> “ This is an example of a quotation. [X, p. Y] ”

References are simply references to sections, figures, code snippets, or chapters written elsewhere in the report. This could look like the following:

This is explained in section X.Y.

Code examples are written in a special environment so that they are easy to read and recognize. Examples can be seen in code snippet 1 and code snippet 2. Whenever there is a sequence of three dots (“...”) in a code snippet, it means that we have omitted some content, which is not important in that specific context.

```
1 class HelloWorldApp
2 {
3     public static void main(String[] args) {
4         System.out.println("Hello, World!"); // Display the string.
5     }
6 }
```

Code snippet 1: *Code example of a hello world program written in Java.*

```
1  bool setup()
2  {
3      println("Hello, World!");
4      return false;
5  }
6
7  bool action()
8  {
9      return false;
10 }
```

Code snippet 2: *Code example of a hello world program written in Arongadongk.*

Source code written in-line is formatted in accordance to what it represents. The following shows an example of a function used in-line:

Code snippet 1 shows the `main` method.

C-Family refers to a category of languages, when used in the report. The C-family contains languages such as: `C`, `C++`, and `Java` [9] [3].

Contents on CD

The CD contains:

Full source code: The full source code of our Arongadongk compiler.
An Arongadongk program: An example program written in Arongadongk.
Report: The report in pdf-version.

Contents

List of Figures

List of Code Snippets

Chapter 1

Prologue

This chapter presents our motivation for this project as well as the criteria for the language and compiler, which we want to design. It contains research on the subject of multi agent systems. A problem statement is defined based on this research, the criteria, and our motivation.

1.1 Motivation

All programming languages have their strengths and weaknesses. When making a programming language, it is important to know what its purpose is. The design process of a programming language includes several decisions, which will determine the language's strengths and weaknesses. One reason to design a new programming language is to simplify the coding of certain programs.

We are fascinated by multi agent systems [11, p. 9], which can be used to simulate and test hypotheses. It gives the programmer tools to learn about behavioral patterns and test the efficiency of an algorithm in a specific environment.

When creating a multi agent system the programmer has to keep track of both agents and the graphical user interface. We do not think that the programmer should spend time on trivial tasks such as setting up a basic environment. We want the programmer to be able to focus on controlling agents and leave as much as possible to the programming language and the compiler. This is why we want to create a language and a compiler that eases the creation of multi agent systems. We want to supply functionality in our language that makes it easy to start the creation of multi agent systems.

We will name our language Arongadongk.

1.2 Multi Agent System

The following quotation explains the goal of a multi agent system:

> “
>
> The goal of multiagent systems' research is to find methods that allow us to build complex systems composed of autonomous agents

> who, while operating on local knowledge and possessing only limited abilities, are nonetheless capable of enacting the desired global behaviors. [11, p. 9]
>
> ”

This quote explains that multi agent systems can be used to figure out which local rules to apply in order to enact the desired global behavior. However, situations may arise when it is of interest to find out what global behavior one would get if applying a specific set of local rules to a set of agents.

1.2.1 Characteristics

There is a set of fundamental characteristics [11, chap. 1] for multi agent systems. These characteristics are the following:

- **Autonomy**: Agents are not controlled by another party. They may still be influenced by their environment, but it is the agent's own functionality that determines its actions.
- **Limited abilities**: A system should not contain an agent that is able to simulate the desired environment on its own because it would defeat the purpose of having multiple agents and thereby having a multi agent system.
- **Decentralization**: There exists no central agent that controls the other agents.

A multi agent system can be used as a tool to research the behavior of each individual agent and give the researcher an understanding of why and how systems work as they do. A multi agent system can also be used to solve a complex problem by defining less complex agents and making them solve it together.

1.2.2 Mechanisms

Beyond the characteristics expressed in the previous section, it is possible to generalize some frequently used concepts within multi agent systems. In this section various mechanisms and their uses will be explained [11, chap. 8].

1.2.2.1 Voting

If agents have local awareness, there may arise situations when it is necessary that the locally aware agents make a mutual decision.

In such situations a voting mechanism allowing the agents to vote on their favorite choice would be attractive.

An example is a situation when one or more “hunters” have a prey surrounded, and an attack from one hunter would interfere with an attack from another hunter, and therefore the hunters need to decide which one should deliver the attack.

1.2.2.2 Auctions

In short, an auction is a simple way of allocating resources among bidding agents. Auctions might also be useful where agents do not have a critical need for a specific resource but still want it, and can therefore choose to bid low.

1.2.2.3 Learning

There are some scenarios where it would be interesting to allow agents to learn. An example could be agents being used to explore an environment and remembering some parts of the environment in an internal map to allow navigation later on.

Cooperative Learning Taking the previous example even further, agents might share their local knowledge with other agents in order to quickly explore the whole environment thereby complete their mutual task faster.

1.3 Criteria

This section explains the criteria for our language and compiler. We will prioritize our criteria in order to argue for our choices later. A brief description of how we want to accomplish our language criteria is found in this section. The criteria presented will be used to evaluate our language in the end of this report.

1.3.1 Defining the Criteria

A programming language can have several different criteria. The criteria which we will base our language choices on are: Writability [7, sec. 1.3.2], readability [7, sec. 1.3.1], orthogonality [7, sec. 1.3.1.2], reliability [7, sec. 1.3.3], cost [7, sec. 1.3.4], and implementability. These are defined below:

Writability describes the effort needed to write a program in a language – the less effort needed, the higher level of writability.

Readability describes the effort needed to read and understand a program written in a language.

Orthogonality means that a language has few constructs, which can be combined in any way. Furthermore, each construct's effect is unique compared to the other constructs.

Reliability is a measure of how much effort is needed by the programmer using a language to make a reliable program, meaning more errors are caught at compile time rather than run time. The less effort needed by the programmer, the higher level of reliability.

Cost is the amount of time needed to use our language. This could be the time it will take for a programmer to learn our language or the time it takes to compile and/or run a program written in our language.

Implementability describes how easy it is to implement a chosen compiler for the language.

These criteria are, by nature, subjective, which means that one programmer may find some code completely unreadable and counterintuitive, while another programmer finds the code readable and intuitive.

1.3.2 Prioritizing Criteria

The language that we want to create must be able to make a simulation containing different agents, i.e. a multi agent system. Since we are making a language and a compiler as part of an educational project we have a strict deadline. This has lead us to prioritize the implementability criterion as the most important.

The language is supposed to enable a programmer to write a multi agent system easily as stated in section 1.1. To accommodate this we have chosen to give writability the second highest priority.

Furthermore, a programmer must be able to read code written in our language, because he may want to use existing code in order to ease the creation of a multi agent system. Therefore, we want to prioritize readability just below writability with third highest importance.

Criterion	Achieved by
Implementability	Limiting language features
Writability	Making syntax and semantics similar to existing languages
Readability	Making syntax and semantics similar to existing languages
Reliability	Introducing types
Cost	Making syntax and semantics similar to existing languages
Orthogonality	Making language constructs general

Figure 1.1: *The criteria in prioritized order with the most important criterion first.*

The fourth most important criterion for our language is the reliability criterion. This criterion is difficult to accomplish if we want the implementability to be important, since more reliability usually means more work on the implementation. Though we believe that this criterion is important because if a multi agent system crashes at runtime a lot of data could be lost.

We want the cost of our language to be the fifth most important criterion. We want it to be easy to learn because it is supposed to make the programming of a multi agent system easier.

The least important criterion for Arongadongk is orthogonality. We have decided that the other criteria should take precedence over this. We consider it to be of some importance, mainly because we want it to be implementable – the fewer constructs, the smaller implementation of the compiler is likely to be needed.

A list ordered by priority is seen in figure 1.1.

1.3.3 Achieving the Criteria

This section explains the actions we will take to achieve the language criteria described and prioritized above. To achieve our most important criterion, implementability, we will start by making a streamlined language with only the necessary features. If sufficient time is available once the necessary feature have been implemented, we will implement additional features.

The writability criteria is achieved by making our language's syntax and semantics similar to those of major languages such as Java and C [1]. The same applies for readability and cost. If a new language is similar to existing languages, we believe programmers are more likely to learn the language quickly and be able to read code written in the language.

In order to increase reliability we will introduce types. By doing this many errors can be detected at compile time rather than run time.

In order to have orthogonality in our language we will try to make every construct as general as possible. We still prioritize implementability higher than orthogonality. This means that some combinations of constructs are not allowed, if the language and its appurtenant functionality should remain implementable in our given time frame.

These criteria will be evaluated at the end of the report.

1.4 Problem Definition

In section 1.2.1 we learn that there are characteristics, which must be fulfilled when building a multi agent system. Therefore, we want to make it as easy as possible to implement these concepts with our language and compiler.

Given our criteria we have created the following preliminary problem statement:

How can we design a programming language and implement a compiler, which allows the programmer to develop his desired multi agent system with as much ease as possible?

1.4.1 Limitations

As explained in the Prioritizing Criteria section (1.3.2) this is an educational report and we have a strict deadline. We will therefore limit ourselves to make a sequential imperative language. We will also – for the sake of implementability – strive to make our language's syntax similar to the C-family, as explained in the Achieving the Criteria section (1.3.3).

We cannot ease the process of creating a multi agent system to the greatest extent due to our deadline.

We want to limit ourselves to make a basic graphical interface and a control structure, which sequentially iterates through each logical step of the program, as we limit ourselves to make a sequential language.

1.4.2 Problem Statement

Based on our preliminary problem statement and our limitations in the previous section, we have chosen the following problem statement:

> *Given our time frame, how can we design a programming language and implement a compiler that relieves the programmer from trivial development tasks when creating a multi agent system?*

Chapter 2

Preliminary Choices

This chapter describes the preliminary choices that need to be made when designing a language and an appurtenant compiler. The language design choices concern types, operators, control structures, and standard environment. The choices we have made prior to implementing the compiler are about choosing a target language and the number of passes the compiler should perform.

2.1 Language Choices

For the types we first have to decide whether we want types, to which we conclude that types suit our wishes best, since it will increase the reliability of our language. To choose which types we want, we refer to our criteria for Arongadongk seen in section 1.3.2. Since implementability is the most important, we will prefer a small number of types, since more types means more work on the implementation. To meet the writability and readability criteria we will give our types names that are similar to those of the C-family. We deem that the types **int** (integer), **float** (real number), **string** (text string), and **bool** (boolean) are sufficient.

Another important decision is which operations we allow for our types. We want operations for our types to be similar to those of the C-family to heighten the readability and writability. We want logical expressions to allow our different control structures to make a choice based on a combination of variables and constants. Comparison is an important expression type because agents might want to compare values when voting or auctioning as mentioned in section 1.2.2. We want arithmetic expressions since these improve writability. The expressions should allow the usage of addition, subtraction, multiplication, and division.

To enable a high level of writability we allow the different kinds of expression to be intertwined. As an example a programmer using Arongadongk is allowed to add the values of two variables and compare the result with a constant value in a single expression.

Since we allow different kinds of expressions to be mixed, we want to define a precedence order, which specifies the order of which different expressions are evaluated. The precedence order for operators in Arongadongk is listed below in ascending order:

- Logical operators: `||` and `&&`

- Unary logical operator: `!`
- Comparison operators: `<`, `>`, `<=`, `>=`, and `==`
- Addition operators: `+` and `-`
- Unary arithmetic operator: `-`
- Multiplication operators: `*` and `/`

We decide that control structures in Arongadongk should enable branching and loops. To enable branching we want to implement if-else statements, an if statement can however also exist without an else statement. Concerning loops we choose to only implement while-loops, since for-loops can be simulated by while-loops.

In our control structures we will use expressions of the boolean type to determine how control should flow. As an example a while-statement will run its body as long as the condition, which is a boolean expression, evaluates to true.

We allow the programmers to define their own functions. The programmer has to define at least two functions, namely `setup` and `action`. The purpose of these functions is described in section 5.2.

2.2 Compiler Structure

A compiler roughly consists of three phases [12, Section 3.1]: A syntactic analysis, contextual analysis, and code generation seen in figure 2.1. The syntactic analysis checks the source code by analyzing each token and constructing a parse tree. A token is an atomic string consisting of valid characters. The contextual analysis takes the parse tree, analyzes it, and decorates the parse tree. The code generation translates the decorated parse tree into the target language. We have to choose the target language and the number of passes the compiler should do, which will be done in the following sections.

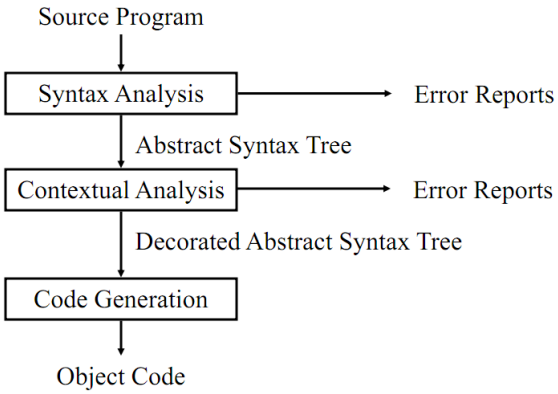


Figure 2.1: *A compiler's three phases.*

2.2.1 Tombstone Diagram

Before the process of writing a compiler can begin, it is necessary to decide the source and target language of the compiler. This can be illustrated using a tombstone diagram.

The Source Language is Arongadongk. Source code written in Arongadongk will be translated into the target language by the Arongadongk compiler.

The Target Language is Java [8], however, the compiler will invoke the Java compiler, so the end result is Java bytecode. We choose Java because it is a high level language, which we all have experience with, thus making it easier for us to implement.

Java is a cross platform object-oriented programming language, which is ideal for us, because we develop on multiple platforms. Java is a widely used programming language. It has libraries that can be used to create a GUI. Choosing Java as target language and writing the compiler in Java makes both our compiler and our compiled Arongadongk programs platform independent [6]. The tombstone diagram for Arongadongk can be seen on figure 2.2.

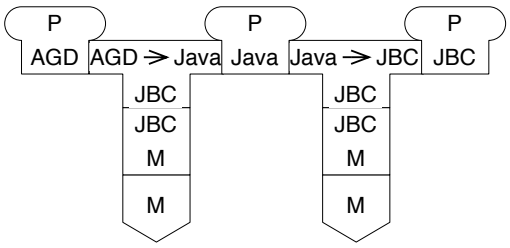


Figure 2.2: *Tombstone diagram for our compiler where the source language is Arongadongk (AGD), the intermediate target language is Java, and the final target language is Java bytecode (JBC).*

2.2.2 Passes

In regards to passes, there are generally two different ways to design a compiler:

One-pass compilation performs the syntactical analysis, contextual analysis, and code generation on the fly [12, Section 3.2.2]. Some advantages are that, in comparison to the multi-pass compiler, the one-pass compiler compiles faster and requires less memory [12, Section 3.2.3]. See figure 2.3 for illustration.

Multi-pass compilation passes, as a minimum, once for the syntactic analysis, the contextual analysis, and the code generation [12, Section 3.2.1], i.e. a minimum of three passes. An advantage of a multi-pass compilation is that it is constructed in a modular way. See figure 2.4 for illustration.

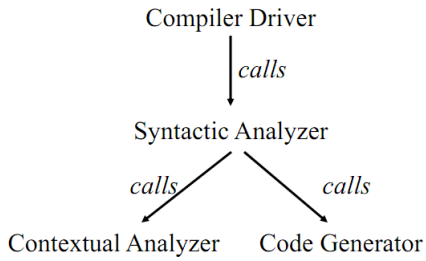


Figure 2.3: *Illustration of a single pass compiler.*

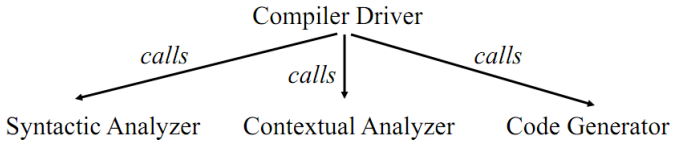


Figure 2.4: *Illustration of a multi pass compiler.*

We believe that the advantages of the multi-pass compiler design outweighs the advantages of the one-pass compiler design. Based on the power of today's computers, we deem that compilation speed and memory usage are not important factors. Factors such as modularity and the fact that we have been taught how the multi-pass compiler design works in the "language and compiler"-course were among the factors that tipped the scale in favor of creating a multi-pass compiler. The modularity could become important if we decide that instead of compiling to Java we compile to another language such as Java bytecode. This would mean our compiler does not need to invoke the Java compiler.

2.2.3 Standard Environment

We choose to have a standard environment, where we want to have our built-in functionality. The functionality we want includes GUI modification and general functions such as a random number generator. The implemented functionalities are described in section 5.2.

Chapter 3

Syntax

This chapter defines the grammar of Arongadongk as well as the first part of the compiler known as syntactic analysis. The syntactic analysis is implemented through a parser, which uses a scanner to convert the source code into tokens. As the parser gets tokens from the scanner it determines the code structure. This structure is then represented as an abstract syntax tree.

3.1 Context Free Grammar

The grammar is written in Backus-Naur Form (BNF) [7, sec. 3.3.1] and defines the syntax of our language, Arongadongk. This section contains the full context free grammar (CFG) for Arongadongk along with an explanation of our choices. This section will also justify why we implement exactly these rules in our CFG.

3.1.1 Backus-Naur Form

BNF is a specific form to write a CFG in. It consists of a set of rules, called production rules, and a set variables, also known as non-terminals. Each production rule consists of a left-hand side containing a variable and a right-hand side describing what the variable evaluates to. A variable can evaluate to new variables, terminals, or a combination of these. Terminals are the valid characters of a language, and are displayed in bold. All variables will eventually evaluate into terminals if the CFG is properly made.

Our CFG is inspired by the Mini-Triangle CFG [12, Appendix B] [13]. The rules are designed to make our language similar to the C-family.

The start symbol is the *program* variable, its production rule is found in rule 3.1. It consists of a *declaration* followed by an end-of-file symbol.

$$\textit{program} ::= \textit{declaration}\ \textbf{EOF} \tag{3.1}$$

3.1.1.1 Declarations

The *declaration* rule consists of a list of *single-declarations*, which means that an Arongadongk program consists of a list of *single-declarations*.

$$\textit{declaration} ::= \textit{single–declaration declaration-sequence} \quad (3.2)$$

$$\textit{declaration-sequence} ::= \textit{single-declaration declaration-sequence} \quad (3.3)$$
$$\mid \varepsilon \quad (3.4)$$

$$\textit{single-declaration} ::= \textit{t-name f-name} \; (\; \textit{formal-parameter-list} \;) \; \textit{single-statement} \quad (3.5)$$
$$\mid \textbf{const } \textit{t-name v-name} = \textit{literal}; \quad (3.6)$$
$$\mid \textbf{global } \textit{t-name v-name}; \quad (3.7)$$

As seen in rules 3.5 - 3.7 we have three different types of *single-declarations*. An example of all three *declarations* used in Arongadongk is shown in code snippet 3.1. Rule 3.5 describes function declarations, which provide the main functionality in Arongadongk. Furthermore, we have rules 3.6 and 3.7, which provide two types of variable declarations not available within functions. They both have a unique keyword to separate them from typical variable declarations. The **global** keyword is specifically made to ease reading. The **const** keyword is also known from other popular programming languages, e.g. C [10].

```
1 const string hello = "Hej";
2 global int currentCount;
3 int countUp() {
4     ...
5 }
```

Code snippet 3.1: *Code example for declarations.*

3.1.1.2 Statements

A function declaration consists of a *single-statement*, which evaluates to one of several specific statements. The block statement in rule 3.10 can be used as a *single-statement* to allow more statements.

Rule 3.13 is three rules in one: A function-call, an assignment, and a variable declaration. They are combined since they have the same starter sets [12, sec. 4.2.4], namely the starter set of *identifier*. The starter set of a production rule is the set of tokens which the rule may start with. Should two production rules for the same variable have starter sets, which overlap, the implementation of the parser has to be more complex. This is described further in section 3.4.1.1. Alternatively, they should have started with *f-name* (function name), *v-name* (variable name), and *t-name* (type name) respectively, but all three evaluate to an identifier.

The way if-statements are handled in the grammar leads to a dangling else problem [7, sec. 3.3.1.10], which we solve by the convention: An *else-statement* belongs to the closest preceding if-statement without an *else-statement*.

$$
\begin{aligned}
\textit{statement} &::= \textit{single-statement}\ \textit{statement} && (3.8)\\
&\mid \varepsilon && (3.9)\\
\\
\textit{single-statement} &::= \textbf{\{}\ \textit{statement}\ \textbf{\}} && (3.10)\\
&\mid \textbf{if (}\ \textit{expression}\ \textbf{)}\ \textit{single-statement}\\
&\quad \textit{else-statement} && (3.11)\\
&\mid \textbf{while (}\ \textit{expression}\ \textbf{)}\ \textit{single-statement} && (3.12)\\
&\mid \textit{identifier}\ \textit{identifier-statement} && (3.13)\\
&\mid \textbf{return}\ \textit{expression}\ \textbf{;} && (3.14)\\
\\
\textit{identifier-statement} &::= \textbf{(}\ \textit{actual-parameter-list}\ \textbf{)\ ;} && (3.15)\\
&\mid \textbf{=}\ \textit{expression}\ \textbf{;} && (3.16)\\
&\mid \textit{v-name}\ \textbf{;} && (3.17)\\
\\
\textit{else-statement} &::= \textbf{else}\ \textit{single-statement} && (3.18)\\
&\mid \varepsilon && (3.19)
\end{aligned}
$$

A function consists of a *single-statement*. We consider it good practice to make it a block statement as expressed in rule 3.10. Furthermore, we want branching, loops, and returns in functions. It is important that the user can test a condition and proceed accordingly, this is why an if-statement is essential and thus included with rule 3.11. Rule 3.18 is added as an intermediate rule because one does not necessarily want to have an *else-statement*. In our opinion it is better even though it causes the dangling else problem, since then the programmer is not required to write an unnecessary *else-statement*.

```
1  int sumUp(int i) {
2      int result;
3      result = 0;
4      if (i < 1) {
5          println("The argument was not valid");
6          return -1;
7      } else {
8          while (i >= 1) {
9              result = result + i;
10             i = i - 1;
11         }
12     }
13     return result;
14 }
```

Code snippet 3.2: *Code example for statements.*

A loop-statement is required to simplify some tasks, e.g. adding numbers from 1-10. We want to be able to receive values from functions. This is done with a return-statement, which is included in rule 3.14. As the rules 3.13 and 3.15 - 3.17 show, one can call a function, initialize a variable, and declare a variable respectively within a function. Examples of these types of statements can be found in code snippet 3.2.

3.1.1.3 Parameters

The parameters are used to parse values from function calls to the body of the function. The parameters given in the declaration are called formal, the parameters given when a function call is invoked are called the actual parameters. With this grammar it is possible to use an arbitrary number of parameters.

$$
\begin{aligned}
\textit{formal-parameter-list} &::= \textit{t-name v-name} \\
&\quad \textit{proper-formal-parameter-list} && (3.20) \\
&\mid \varepsilon && (3.21)
\end{aligned}
$$

$$
\begin{aligned}
\textit{proper-formal-parameter-list} &::= \texttt{,}\ \textit{t-name v-name} \\
&\quad \textit{proper-formal-parameter-list} && (3.22) \\
&\mid \varepsilon && (3.23)
\end{aligned}
$$

$$
\begin{aligned}
\textit{actual-parameter-list} &::= \textit{expression} \\
&\quad \textit{proper-actual-parameter-list} && (3.24) \\
&\mid \varepsilon && (3.25)
\end{aligned}
$$

$$
\begin{aligned}
\textit{proper-actual-parameter-list} &::= \texttt{,}\ \textit{expression} \\
&\quad \textit{proper-actual-parameter-list} && (3.26) \\
&\mid \varepsilon && (3.27)
\end{aligned}
$$

Both lists have a "proper" intermediate rule. These "proper" rules, as found in rule 3.22 and 3.26, are used to add more than one parameter, due to the restrictions of BNF. The formal parameter list only works with the types in our language. The actual parameter list works differently as it can take expressions as parameters, since these will evaluate to a value of a type. The two types of parameter lists can be seen in code snippet 3.3.

```
1  int countDown(int n, int step, string text) {
2      ...
3      return n - step;
4  }
5  bool foo() {
6      int m;
7      m = countDown(5, 2, "hello");
8      m = countDown(countDown(m, 1, "hello"), 3, "world");
9      return true;
10 }
```

Code snippet 3.3: *Code example for parameters.*

3.1.1.4 Expression

The rules here are made so the operator precedence will become as described in the Language Choices section (2.1).

expression	::= *logical-helper unary-logic-expression*	(3.28)
logical-helper	::= *expression logical-operator*	(3.29)
	| ε	(3.30)
unary-logic-expression	::= *unary-logic-operator comparison-expression*	(3.31)
	| *comparison-expression*	(3.32)
comparison-expression	::= *comparison-helper addition-expression*	(3.33)
comparison-helper	::= *comparison-expression comparison-operator*	(3.34)
	| ε	(3.35)
addition-expression	::= *addition-helper unary-arithmethic-expression*	(3.36)
addition-helper	::= *addition-expression addition-operator*	(3.37)
	| ε	(3.38)
unary-arithmetic-expression	::= *unary-arithmetic-operator multiplication-expression*	(3.39)
	| *multiplication-expression*	(3.40)
multiplication-expression	::= *multiplication-helper factor*	(3.41)
multiplication-helper	::= *multiplication-expression multiplication-operator*	(3.42)
	| ε	(3.43)
factor	::= *identifier factor-helper*	(3.44)
	| *literal*	(3.45)
	| (*expression*)	(3.46)
factor-helper	::= (*actual-parameter-list*)	(3.47)
	| ε	(3.48)

The general idea with our expressions are rather straight forward, as we want the precedence explained in the Language Choices section we start with the rules with lowest precedence. The further down we get in the rules the higher precedence the operators have. The "helper" rules 3.29, 3.34, etc. help us achieve the same kind of recursion as the "proper" rules did with our parameter lists. The "proper" rules are found in production rule 3.22.

3.1.1.5 Identifiers

Identifiers must start with a letter, and may be followed by a combination of letters and numbers.

$$\textit{f-name} ::= \textit{identifier} \tag{3.49}$$

$$\textit{v-name} ::= \textit{identifier} \tag{3.50}$$

$$\textit{t-name} ::= \textit{identifier} \tag{3.51}$$

$$\textit{identifier} ::= \textit{letter identifier-helper} \tag{3.52}$$

$$\begin{aligned} \textit{identifier-helper} ::= {} & \textit{letter identifier-helper} && (3.53)\\ | \ & \textit{numeric identifier-helper} && (3.54)\\ | \ & \varepsilon && (3.55) \end{aligned}$$

The three types of identifiers (rule 3.49 - 3.51) are function names, variable names, and type names. In rule 3.52 we see an intermediate rule, *identifier-helper*, which helps us achieve the recursion we want. The *identifier-helper* rule shown in rule 3.53 contains letters and numbers, whereas the *identifier* must start with a letter.

3.1.1.6 Literals

Literals are the possible values in the program. There are four different literals, which are all linked to a type in the contextual analysis. As with the expressions and identifiers we have two "helper" rules. These are *int-float-difference* and *int-helper*, found at rules 3.60 - 3.64. Since a floating point can start with either a number or a period (".") we use the same rule for both integer and floating point. A floating point can be created from either rule 3.56 or 3.57, where integers can only by created from rule 3.56. Booleans are two literals "false" and "true". This infers that Arongadongk has the following four literals: String literals, integer literals, floating point literals, and boolean literals.

An example of literals and comments can be seen in code snippet 3.4. There is a "graphic" literal rule 3.68, which is used within strings and comments.

$$
\begin{aligned}
\textit{literal} &::= \textit{numeric int-float-difference} && (3.56)\\
&\mid \textbf{.}\ \textit{numeric int-helper} && (3.57)\\
&\mid \textit{stringliteral} && (3.58)\\
&\mid \textit{booleanliteral} && (3.59)
\end{aligned}
$$

$$
\begin{aligned}
\textit{int-float-difference} &::= \textit{numeric int-float-difference} && (3.60)\\
&\mid \textbf{.}\ \textit{int-helper} && (3.61)\\
&\mid \varepsilon && (3.62)
\end{aligned}
$$

$$
\begin{aligned}
\textit{int-helper} &::= \textit{numeric int-helper} && (3.63)\\
&\mid \varepsilon && (3.64)
\end{aligned}
$$

$$
\textit{stringliteral} ::= \text{"}\textit{graphic}\text{"} \quad (3.65)
$$

$$
\begin{aligned}
\textit{booleanliteral} &::= \textbf{true} && (3.66)\\
&\mid \textbf{false} && (3.67)
\end{aligned}
$$

$$
\begin{aligned}
\textit{graphic} &::= \textit{sign graphic} && (3.68)\\
&\mid \varepsilon && (3.69)
\end{aligned}
$$

```
1 const int intLiteral = 42;
2 const string stringLiteral = "A literal";
3 const float floatLiteral1 = .66;
4 const float floatLiteral2 = 0.66;
5 /* A comment
6     on several lines
7 */
8 // Single line comment
```

Code snippet 3.4: *Code example for literals.*

3.1.1.7 Terminals

The terminals are the basics of the language. A terminal can only evaluate to a token. They define the allowed characters to be used.

logical-operator ::= **&&** | **||** (3.70)

unary-logic-operator ::= **!** (3.71)

comparison-operator ::= **<** | **>** | **>=** | **<=** | **==** | **!=** (3.72)

addition-operator ::= **+** | **−** (3.73)

unary-arithmetic-operator ::= **−** (3.74)

multiplication-operator ::= ***** | **/** (3.75)

numeric ::= **0** | ... | **9** (3.76)

letter ::= **a** | ... | **z** | **A** | ... | **Z** (3.77)

comment ::= **/*** *comment-content* ***/** (3.78)
| **//** *graphic* **EOL** (3.79)

comment-content ::= *sign comment-content* (3.80)
| **EOL** *comment-content* (3.81)
| **LF** *comment-content* (3.82)
| ε (3.83)

sign ::= [all utf8 characters except for end-of-line, line-feed, quote, and end-of-file] (3.84)

The only rules that are not straight forward are 3.78, 3.79, and 3.84. The first two rules – the comments – describe the way to add comments in our language. They are very similar to the comments in the C-family. There are two options for comments: Multiple line and single line. A multiple line comment is initiated with **"/*"** and ended with **"*/"** and it can contain any character within it. A single line is slightly different as it is initiated with **"//"** and ends with an **"EOL"** character (end-of-line). Between **"//"** and **"EOL"** it can contain any character in the *sign* rule. The sign rule at rule 3.84 contains every single character available in utf8, except for the quotation sign ("), **"EOL"**, **"LF"** (line-feed), and **"EOF"**.

3.2 Extended Backus-Naur Form

In order to make the CFG from section 3.1 easier to implement, we use an extension of the BNF. The grammar still describes the same language just using a slightly different syntax. This section presents our grammar written in Extended BNF (EBNF [7, sec. 3.3.2]) and will only focus on the changes compared to the grammar written in BNF, and give an explanation of those changes. When converting a grammar written in BNF to EBNF the rules are simplified using regular expressions. Everything that has not changed from the BNF to the EBNF is not listed.

3.2.1 Declarations

We use the Kleene cross "+" to indicate that there are one or several occurrences of an element. The *declaration-sequence* rule (3.3) in the grammar in BNF is no longer needed because of the recursive construct in rule 3.86. Furthermore, we have made changes to the *single-declaration* rule. It no longer distinguishes between the different types of identifiers as they are all identifiers to the parser. The different types of identifiers in the grammar in BNF can be seen in section 3.1.1.5.

$$\textit{program} ::= \textit{declaration}\ \textbf{EOF} \tag{3.85}$$

$$\textit{declaration} ::= \textit{single-declaration}^{+} \tag{3.86}$$

$$\begin{aligned}\textit{single-declaration} ::= {} & \textit{identifier}\ \textit{identifier}\ (\ \textit{formal-parameter-list}\ ) \\ & \textit{single-statement} && (3.87)\\ \mid {} & \textbf{const}\ \textit{identifier}\ \textit{identifier}\ =\ \textit{literal}\ ; && (3.88)\\ \mid {} & \textbf{global}\ \textit{identifier}\ \textit{identifier}\ ; && (3.89)\end{aligned}$$

3.2.2 Statements

Statement rules we can eliminate are: *identifier-statement* (rule 3.13) on page 23 and *else-statement* (rule 3.18). It is not necessary to have a recursive call to *statement* in the *statement* rule 3.8, as we can simply indicate, that we have zero or more *single-statements*, using the Kleene star, see rule 3.90.

$$\textit{statement} ::= \textit{single-statement}^{*} \tag{3.90}$$

$$\begin{aligned}\textit{single-statement} ::= {} & \textbf{if}\ (\ \textit{expression}\ )\ \textit{single-statement} \\ & (\ \textbf{else}\ \textit{single-statement}\ \mid \varepsilon\ ) && (3.91)\\ \mid {} & \textbf{while}\ (\ \textit{expression}\ )\ \textit{single-statement} && (3.92)\\ \mid {} & \textit{identifier}\ (\ (\ \textit{actual-parameter-list}\ )\ ; \\ & \mid =\ \textit{expression}\ ;\ \mid\ \textit{identifier}\ ;\ ) && (3.93)\\ \mid {} & \textbf{return}\ \textit{expression}\ ; && (3.94)\end{aligned}$$

3.2.3 Parameter Lists

The two parameter list rules 3.20 - 3.23 and 3.24 - 3.27 on page 24 from the grammar in BNF are so similar that the following applies for both: The rules t-name and v-name have been replaced with identifiers because they evaluate to identifiers as seen in section 3.1.1.5 on page 26. Since we can use the Kleene star we do not need to have the proper parameter list rules (i.e. *proper-formal-parameter-list* rules 3.22 - 3.23 and *proper-actual-parameter-list* rules 3.26 - 3.27) as in the grammar in BNF, thus they have been removed. The result can be seen in rule 3.95 - 3.98.

$$\begin{aligned} \textit{formal-parameter-list} ::= {} & \textit{identifier identifier} \; (\; \textbf{,} \; \textit{identifier identifier} \;)^* && (3.95) \\ \mid {} & \varepsilon && (3.96) \end{aligned}$$

$$\begin{aligned} \textit{actual-parameter-list} ::= {} & \textit{expression} \; (\; \textbf{,} \; \textit{expression} \;)^* && (3.97) \\ \mid {} & \varepsilon && (3.98) \end{aligned}$$

3.2.4 Expressions

The expression rules are all similar. A general conversion rule is to remove the "helper" rules, see rules 3.28 on page 25 - 3.48 on page 25. All the "helper" rules are moved into the rules in which they are used, by using the Kleene star.

The *expression* rule (3.99), which instead of containing a *logical-helper* rule (3.29) and a *unary-logical-expression* rule (3.31) contains a *logical-expression* rule (3.100).

The rules that are removed in the grammar in EBNF are: *logical-helper* (3.29), *comparison-helper* (3.34), *addition-helper* (3.37), *multiplication-helper* (3.42), and *factor-helper* (3.47). On the right side of *factor* rule (3.44) *factor-helper* is removed but the remaining rules 3.47 - 3.48 are not changed.

$$\textit{expression} ::= \textit{logical-expression} \quad (3.99)$$

$$\begin{aligned} \textit{logical-expression} ::= {} & \textit{unary-logic-expression} \; (\; \textit{logical-operator} \\ & \textit{unary-logic-expression} \;)^* && (3.100) \end{aligned}$$

$$\begin{aligned} \textit{unary-logic-expression} ::= {} & (\; \textit{unary-logic-operator} \mid \varepsilon \;) \\ & \textit{comparison-expression} && (3.101) \end{aligned}$$

$$\begin{aligned} \textit{comparison-expression} ::= {} & \textit{addition-expression} \; (\; \textit{comparison-operator} \\ & \textit{addition-expression} \mid \varepsilon \;) && (3.102) \end{aligned}$$

$$\begin{aligned}
&\textit{addition-expression} && ::= \textit{unary-arithmetic-expression}\ (\ \textit{addition-operator} \\
&&& \quad \textit{unary-arithmetic-expression}\ )^* && (3.103)
\end{aligned}$$

$$\begin{aligned}
&\textit{unary-arithmetic-expression} && ::= (\ \textit{unary-arithmetic-operator} \mid \varepsilon\ ) \\
&&& \quad \textit{multiplication-expression} && (3.104)
\end{aligned}$$

$$\textit{multiplication-expression} \quad ::= \textit{factor}\ (\ \textit{multiplication-operator}\ \textit{factor}\ )^* \qquad (3.105)$$

$$\begin{aligned}
&\textit{factor} && ::= \textit{identifier}\ (\ \textbf{(}\ \textit{actual-parameter-list}\ \textbf{)}\ \mid \varepsilon\ ) && (3.106) \\
&&& \mid\ \textit{literal} && (3.107) \\
&&& \mid\ \textbf{(}\ \textit{expression}\ \textbf{)} && (3.108)
\end{aligned}$$

3.2.5 Identifiers & Literals

As explained in section 3.2.3 we no longer differentiate between the different identifiers thus the rules for them have been removed. The removed rules are: *f-name* (3.49), *v-name* (3.50), and *t-name* (3.51) on page 26. The *identifier* rules 3.52 - 3.55 have been changed into rule 3.116 by using a Kleene star construction.

The *literal* rules (3.56 - 3.59) in section 3.1.1.6 has four new production rules, see rules 3.109 - 3.112. The *int-float-difference* is now expressed using regular expressions, therefore, the rules 3.60 - 3.62 have been replaced with rule 3.114. There is now a separate *intliteral* rule 3.113 which uses the Kleene cross for the recursion. The *stringliteral* rule 3.58 has not changed nor has the *booleanliteral* rule.

All Terminal rules and operator rules stay the same from the grammar in BNF to the grammar in EBNF, they can be seen in section 3.1.1.7.

$$\begin{aligned}
&\textit{literal} && ::= \textit{intliteral} && (3.109) \\
&&& \mid\ \textit{floatliteral} && (3.110) \\
&&& \mid\ \textit{stringliteral} && (3.111) \\
&&& \mid\ \textit{booleanliteral} && (3.112)
\end{aligned}$$

$$\textit{intliteral} \quad ::= \textit{numeric}^+ \qquad (3.113)$$

$$\begin{aligned}
&\textit{floatliteral} && ::= \textit{numeric}^*\ \textbf{.}\ \textit{numeric}^+ && (3.114) \\
&&& \mid\ \textit{numeric}^+\ \textbf{.} && (3.115)
\end{aligned}$$

$$\textit{identifier} \quad ::= \textit{letter}\ (\ \textit{letter} \mid \textit{numeric}\ )^* \qquad (3.116)$$

$$\textit{graphic} \quad ::= (\ \textit{letter} \mid \textit{numeric} \mid \textit{sign} \mid \textit{white-space}\ )^* \qquad (3.117)$$

3.3 Scanner

The purpose of the scanner is to translate the Arongadongk source code from a stream of characters into tokens, which the parser can parse.

This section describes how our scanner is implemented using the EBNF described in section 3.2.

3.3.1 Scanner Design & Implementation

The general structure of the scanner is inspired by the one of Mini Triangle [12, appendix B], i.e. the Arongadongk compiler's scanner has the same functions as the Mini Triangle scanner.

The implementation of tokens is done through the **Token** class. The class has the following three fields:

Spelling is how the given token is spelled.

Type is one of the kinds seen in code snippet 3.5.

Line Number represents which line the token is found in the source code.

```
1  public enum TokenKind {
2      IDENTIFIER,
3      INTLITERAL,
4      FLOATLITERAL,
5      STRINGLITERAL,
6      LEFTPARENTHESIS,
7      RIGHTPARENTHESIS,
8      LEFTCURLYBRACKET,
9      RIGHTCURLYBRACKET,
10     SEMICOLON,
11     COMMA,
12     LOGICALOPERATOR,
13     UNARYLOGICALOPERATOR,
14     COMPARISONOPERATOR,
15     ADDITIONOPERATOR,
16     MULTIPLICATIONOPERATOR,
17     ASSIGNMENT,
18     CONST,
19     RETURN,
20     GLOBAL,
21     IF,
22     ELSE,
23     TRUE,
24     FALSE,
25     WHILE,
26     EOF;
27     ...
28 }
```

Code snippet 3.5: *Content of the TokenKind enumeration.*

Apart from the constructor the scanner only has one public method – the `scan` method. The `scan` method is called by the parser to scan for the next

Token in the Arongadongk source code. When `scan` is called it calls the `scanToken` method. The `scanToken` method consists of a number of if-else statements, which are used to determine the type of the token. Depending on this, `scanToken` either returns the token or keeps appending until it reaches a separator e.g. a white space. An example would be if it reads a "(" it will immediately determine that the token is a left parenthesis. After each token has been processed it is either returned or an exception is thrown. The latter only occurs in case the input from the stream does not match a token type in the language. The `scanToken` method uses the methods `isDigit` and `isLetter` to determine the type of characters in the stream.

The spelling of the token is now saved in the string buffer **currentSpelling**, and the type of the token has been returned to the `scan` method by the `scanToken` method. The `scan` method then initializes a **Token** object of the given token type with spelling and line number and returns it to the parser.

3.4 Parser

A parser is the component of the compiler that uses the tokens provided by the scanner to populate a phrase structure, which in our case is an abstract syntax tree (AST). The AST is explained in section 3.5.

The parser utilizes the language's grammar from section 3.2 to check whether the provided source code has a valid syntax. If the syntax is invalid it is also the parser's responsibility to provide sensible error messages, thereby making it easier for the programmer to correct the syntactical error.

Notice that the parser is based on the grammar in the Extended Backus-Naur Form section.

3.4.1 Designing a Parser

There are a number of choices that have to be made when creating a parser. A central one being how to construct the AST. There are many different types of parsers, but there are two main categories for parsers: Bottom-up parser and top-down parser [7, p. 200]. The difference between a top-down and a bottom-up parser is the direction which the syntax tree is being constructed.

A Bottom-Up parser examines the input tokens from left to right. Whenever a sub-tree can be created from the examined tokens the parser does so. Before creating a sub-tree the parser analyzes the previously made nodes, and determines which node is best suited in that particular context. By doing this the syntax tree is created bottom-up [12, sec. 4.3.1]. This is illustrated in figure 3.1.

A Top-Down parser creates a syntax tree by following the rules of the grammar from the start symbol, see EBNF rule 3.85. It compares the rules of the grammar with the input tokens from left to right. If the grammar yields a non-terminal symbol, the same method is used for that symbol's production rule. This way the syntax tree is created top-down [12, sec. 4.3.2]. This is illustrated in figure 3.2.

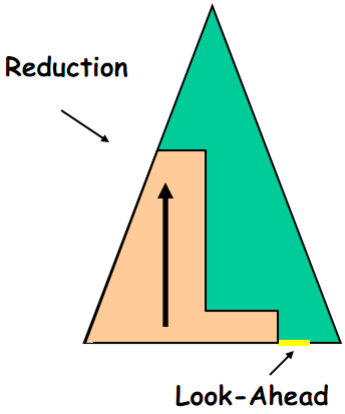


Figure 3.1: *The Bottom-Up method.*

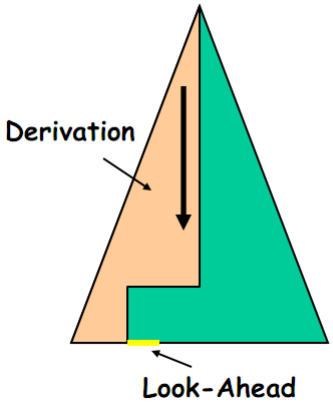


Figure 3.2: *The Top-Down method.*

We choose to use the top-down algorithm because we learned about it in the "language and compilers"-course, which makes it easy for us to use and implement.

3.4.1.1 Recursive-Descent Parsing

We choose the recursive-descent parser – which is a top-down based parser – as it is easy to implement due to the fact that it is covered in the "language and compilers"-course. When a recursive-descent parser encounters a non-terminal, it must choose between its production rules. For it to choose the correct one, it will have to examine a number of tokens ahead. A grammar is said to be $LL(k)$ if a recursive-descent parser has to examine k tokens ahead when determining which production rule to use. The first L of $LL(k)$ specifies that the scanning of input is from left to right. The second L specifies that the syntax-tree is derived from the left [7, p. 202].

It is desirable to keep the k-value low for the sake of simplicity. Specifically $LL(1)$ is a property we would like to have for our language. The grammar seen in section 3.1 is $LL(1)$. The reason for the low k-value is that it should keep our implementation simple. It is also the case that almost any grammar for a programming language can be transformed into an $LL(1)$ without changing the language it generates [12, p. 104].

3.4.2 Implementing a Parser

The parser is represented as a class in our implementation of the Arongadongk compiler. The **Parser** class has a number of methods of which three are public; the constructor, the `parse` method, and the `printErrors` method. The **Parser** has a private method for each non-terminal X (these non-terminals are found in section 3.2) called `parseX`. It is the responsibility of each of these methods to create an X node for the AST and return it. The method `parse` gets a new token from the scanner and calls the method `parseProgram`. It then ensures that there is no more source code after the program, i.e. that the end of the file has been reached. The **Parser** class has a list, named errors, to which

all syntactical errors are added. The method `printErrors` utilizes this list to display the errors in the Arongadongk source code it tried to parse.

```
1  private Declaration parseDeclaration() throws IOException {
2      ...
3      Declaration d1AST = parseSingleDeclaration();
4      while (currentToken.getKind() != TokenKind.EOF)
5      {
6          SingleDeclaration d2AST = parseSingleDeclaration();
7          d1AST = new MultipleDeclaration(d1AST, d2AST);
8      }
9      return d1AST;
10 }
```

Code snippet 3.6: *The source code for the `parseDeclaration` method.*

In code snippet 3.6 the method that parses a declaration is shown. As seen in section 3.2.1 a program consists of a declaration and an **EOF** character, and that declaration consists of one or more single declarations. Line 3 shows that a single declaration is unconditionally parsed and saved in the declaration d1AST. In the while loop on line 4, which runs until the end of the file has been reached, a new single declaration is parsed and saved in the single declaration d2AST. A multiple declaration, which syntax is seen in section 3.2.1, is created with all the single declarations in d1AST and the newly parsed single declaration in d2AST. This is saved in the declaration d1AST. Every time a single declaration is parsed the method `parseSingleDeclaration` is called, which is shown in code snippet 3.7.

In code snippet 3.7 the source code for the method, which parses a single declaration is shown (see the Declarations section for the single declaration production rule). A single declaration can be parsed as a function declaration, constant declaration, or global variable declaration cf. production rules 3.87, 3.88, or 3.89 respectively. As seen on line 5 if the current token is an identifier, it will call methods needed to parse a function declaration. A function declaration consists of a type, a name, a left parenthesis, a parameter list, a right parenthesis, and a single statement. The type and name are both identifiers, so the `parseIdentifier` function is called to parse these. The `accept` method is called on a left parenthesis, which scans a new token and adds any potential errors to the error list. The formal parameters are parsed by the `parseFormalParameterList`. The right parenthesis is accepted like the left one. Then the body of the function is parsed by the `parseSingleStatement` method. Finally the **SingleDeclaration** is created using all the parsed input.

If the `parseSingleDeclaration` method is called while the current token is "CONST", it is assumed that the single declaration is a constant declaration, and if the current token is "GLOBAL", it is assumed that the single declaration is a global variable declaration. The source code for parsing and creating such single declarations is omitted in this section for the sake of brevity.

If the current token is not an identifier, "CONST", or "GLOBAL", an error with the expected tokens is added to the error list. In the end of the function the single declaration is returned.

The method `parseSingleDeclaration` is an example of a parsing method, which determines the kind of sentence that should be parsed.

```
1  private SingleDeclaration parseSingleDeclaration() throws
       IOException
2  {
3      ...
4      SingleDeclaration sAST = null;
5      if (currentToken.getKind() == TokenKind.IDENTIFIER)  {
6          Identifier type = parseIdentifier();
7          Identifier name = parseIdentifier();
8          accept(TokenKind.LEFTPARENTHESIS);
9          FormalParameterList formalParameters =
               parseFormalParameterList();
10         accept(TokenKind.RIGHTPARENTHESIS);
11         SingleStatement body = parseSingleStatement();
12         sAST = new FunctionDeclaration(type, name, formalParameters
               , body);
13     }
14     else if (currentToken.getKind() == TokenKind.CONST)
15     { ... }
16     else if (currentToken.getKind() == TokenKind.GLOBAL)
17     { ... }
18     else {
19         ArrayList<TokenKind> expected = new ArrayList<TokenKind>();
20         expected.add(TokenKind.IDENTIFIER);
21         expected.add(TokenKind.CONST);
22         expected.add(TokenKind.GLOBAL);
23         expected.add(TokenKind.EOF);
24         errors.add(new UnexpectedTokenExpectedRangeException(
               expected, currentToken));
25         acceptIt();
26     }
27     return sAST;
28 }
```

Code snippet 3.7: *The source code for the `parseSingleDeclaration` method. Code for parsing constants and global variables has been omitted.*

3.5 Abstract Syntax Tree

We are using an abstract syntax tree in our compiler to represent the structure of a program and it will be presented in this section. It is important to define our abstract syntax tree because it is an essential part of the recursive-descent parser, which our compiler is using. The abstract syntax tree is based on the context free grammar in BNF defined in section 3.1.

3.5.1 Defining an Abstract Syntax Tree

In this section the abstract syntax tree is defined and what makes a syntax tree abstract. A syntax tree illustrates a derivation from a context free grammar. To derive basically means to replace non-terminals with their production rules until there are only terminals left. During the derivation each non-terminal and terminal is saved as a node of the syntax tree, which is being grown.

Every time a non-terminal is replaced with the right hand side of a production rule, the terminals and non-terminals of the right hand side of the production rule become children of the non-terminal being replaced. The root of the syntax tree is the start symbol of the context free grammar. An example

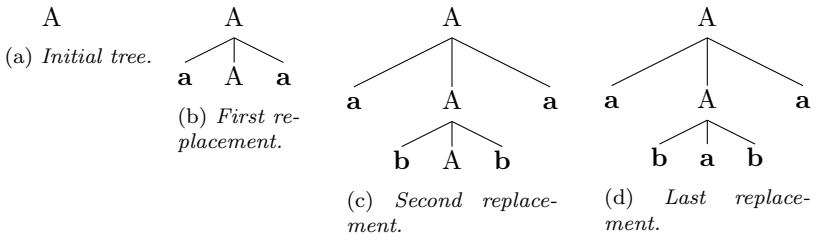


Figure 3.3: *A syntax tree derived from the context free grammar 3.118. The first, second, and third production rules are used to generate the trees in 3.3b, 3.3c, and 3.3d respectively.*

of a syntax tree can be seen in figure 3.3 where the context free grammar in production rule 3.118 is used to derive it in the following order: The start symbol A is replaced with **a** A **a** according to the first production rule. The new A is then replaced with **b** A **b** according to the second production rule. Lastly the newest A is replaced with **a** using the third production rule. This derivation yields the string **ababa**.

$$A ::= \mathbf{a}\ A\ \mathbf{a} \mid \mathbf{b}\ A\ \mathbf{b} \mid \mathbf{a} \mid \mathbf{b} \mid \varepsilon \tag{3.118}$$

A syntax tree is abstract if irrelevant terminals and non-terminals are not saved in the tree. For an example some languages use "`=`" as assignment operator, while other languages use "`:=`" [12, p. 11-15]. In an abstract syntax tree the operator is not saved in the tree because it has no meaning after it has been determined that it is an assignment operator.

The parser in our compiler will make an abstract syntax tree to structure the program being compiled. This is described further in section 3.4.

3.5.2 Designing an Abstract Syntax Tree

We will design the abstract syntax tree by making every non-terminal into an abstract class. Every production rule is then made into a concrete class which inherits from the abstract class [12, sec. 4.4]. Each concrete class will have child nodes according to the non-terminals in the corresponding production rule and each terminal, which does not have a predefined spelling. Figure 3.4a shows how the while-production-rule is represented as an abstract syntax tree. We represent abstract classes in italic and concrete classes in typewriter. Each concrete class will have a subscript showing the abstract base class.

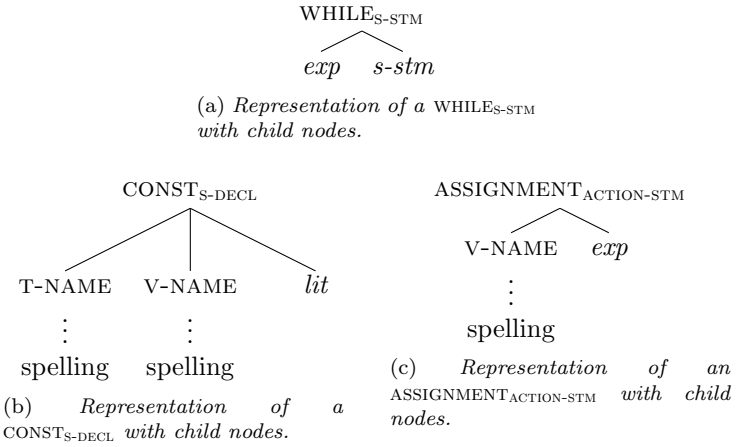
We will combine the production rule 3.13 with all the production rules for *identifier-statement*. This means that the abstract syntax for single statement, which we are making abstract syntax trees for does not have the production rule:

$$statement ::= identifier\ identifier\text{-}statement \tag{3.119}$$

Instead it has the following three production rules added:

$$\textit{statement} ::= \textit{f-name}\ (\ \textit{actual-parameter-list}\ )\ ; \tag{3.120}$$
$$\mid\ \textit{v-name} = \textit{expression}\ ; \tag{3.121}$$
$$\mid\ \textit{t-name}\ \textit{v-name}\ ; \tag{3.122}$$

These production rules are called function call, assignment, and variable declaration, respectively.



(a) *Representation of a* WHILE$_{\text{S-STM}}$ *with child nodes.*

(b) *Representation of a* CONST$_{\text{S-DECL}}$ *with child nodes.*

(c) *Representation of an* ASSIGNMENT$_{\text{ACTION-STM}}$ *with child nodes.*

Figure 3.4: *Examples of the design of our abstract syntax tree.*

The constant declaration is illustrated in figure 3.4b. It shows that *t-name* and *v-name* have spellings. This is because we need to differentiate between different variables and functions, e.g. if the value 5 is assigned to the constant x, it is important that the value 5 is retrieved when the x value is called for.

Figure 3.4c shows how we represent the assign statement.

The action statement is an abstract class. This statement derives into three classes corresponding to production rules 3.120 - 3.122.

The classes **AssignmentStatement**, **DeclarationStatement**, and **FunctionCallStatement** inherit from the abstract class ***ActionStatement***. It makes it resemble the CFG, since every statement derived from production rule 3.119 is now an ***ActionStatement***.

In figure 3.4b T-NAME and V-NAME are actually identifiers, but since they are different fields in the class they have been given different names. The same applies for V-NAME in figure 3.4c.

3.5.3 Implementing an Abstract Syntax Tree

When implementing an abstract syntax tree for Arongadongk we use the classes described in section 3.5.2. We also add an extra abstract class, ***AST***, to be able to give every class in the abstract syntax tree some common methods: `getLocation` and `visit`. The `visit` method is explained in section 3.5.4. The method `getLocation` is used to get the line number in the source file, where the given ***AST*** object is located.

It is worth noting that both declarations and statements have been implemented with left recursion and not right recursion as our context free grammar might suggest.

It is necessary to add a few interfaces to ease the implementation, namely; ***IFunctionCall***, ***IVariableDeclaration***, and ***IDeclaration***. The two latter interfaces are shown in a class diagram in figure 3.5. It is necessary to create these two interfaces due to Java's limited inheritance – a class can only inherit from a single class [5].

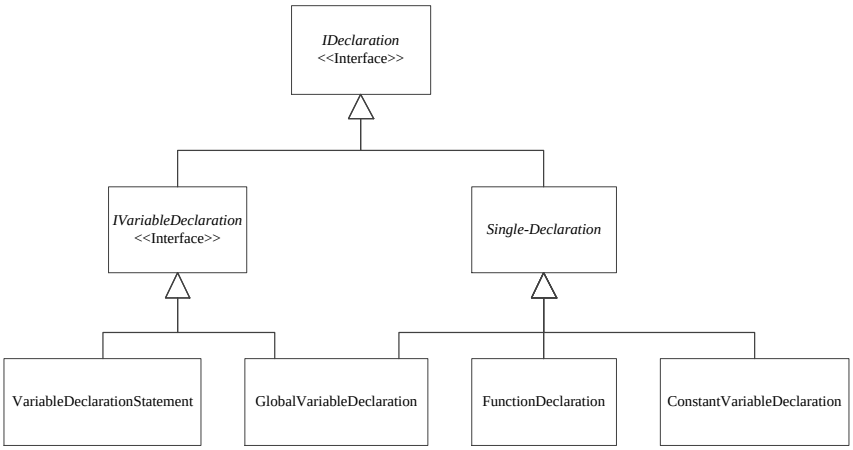


Figure 3.5: *The class diagram showing how* ***IDeclaration*** *is related to the four concrete declaration classes. Note that* ***VariableDeclarationStatement*** *is not a* ***SingleDeclaration****, but a* ***SingleStatement****.*

In section 3.5.2 we see that **VariableDeclarationStatement** is listed with the other single-statements even though it is still a declaration, thus we need something to gather all declarations in the implementation. This is achieved with the interface ***IDeclaration***, but we need to separate it further. As seen in figure 3.5 both **VariableDeclarationStatement** and **GlobalVariableDeclaration** inherits from ***IVariableDeclaration***, since the value of these can be changed later in the program whereas **FunctionDeclaration** and **ConstantVariableDeclaration** cannot.

The interface ***IFunctionCall*** is implemented by classes that call functions in Arongadongk. There are two classes that handle function calls. They are called: **FunctionCallFactor** and **FunctionCallStatement** and are used in factor and as single-statements respectively.

When all the classes are created they need variables to hold the variables described by the BNF rules. These variables are then initialized when an object of the class is constructed. The general rule is that anything that is not in bold in the BNF rules needs to be a variable in the class that corresponds to that particular rule. An example of a class in our abstract syntax tree is the **ConstantVariableDeclaration** class as shown in code snippet 3.8. The BNF rule 3.6 describes that when declaring a constant you need the keyword **const** along with a t-name identifier and a v-name identifier as well as a literal. When

```
1  public class ConstantVariableDeclaration extends SingleDeclaration{
2      public Identifier vType;
3      public Identifier vName;
4      public Literal literal;
5      ...
6      public ConstantVariableDeclaration (Identifier type, Identifier
           name, Literal value) {
7          vType = type;
8          vName = name;
9          literal = value;
10     }
11     ...
12 }
```

Code snippet 3.8: *The **ConstantVariableDeclaration** class in our abstract syntax tree.*

a **ConstantVariableDeclaration** object is constructed it has already been determined that it is a constant declaration, thus it is not necessary to save the keyword **const** in a variable, nor do we need to save the = sign as described in section 3.5.1. This means we only save the t-name identifier, the v-name identifier, and the literal.

3.5.4 Visitor Pattern

A visitor pattern is a way to traverse an abstract syntax tree. A visitor pattern is implemented with the idea that the implementation of multiple visitors is eased, since the implementer only have to write the body of methods to visit every concrete class in the abstract syntax tree.

To traverse an abstract syntax tree you need a method for every concrete class, this method is called `visit`. The `visit` method is an abstract method, which every concrete class needs to implement. An example of a `visit` method can be seen in code snippet 3.9. The `visit` method contains just a single statement; the return statement. The return statement calls the specific method for the given class on the ***IVisitor*** object called **v** and returns the result of that method.

```
1 public Object visit(IVisitor v, Object arg) {
2     return v.visitConstantVariableDeclaration(this, arg);
3 }
```

Code snippet 3.9: *An example of the implementation of the `visit` method in the class **ConstantVariableDeclaration**.*

The class of **v** needs to implement the ***IVisitor*** interface and thereby all the methods defined in it. There are methods in the ***IVisitor*** for every concrete abstract syntax tree class. A few examples of methods in the ***IVisitor*** are: `visitConstantVariableDeclaration`, `visitProgram`, `visitWhileStatement`, and `visitReturnStatement`.

An example of a visitor implementing the `visitWhileStatement` can be seen in code snippet 3.10. The code snippet shows an implementation of

`visitWhileStatement` in the **Checker** visitor that is explained in further detail in section 4.4.1.

```
1  public Object visitWhileStatement(WhileStatement state, Object arg)
       {
2      state.expression.visit(this, null);
3
4      if(state.expression.getType() != StandardEnvironment.
           getBoolTypeDeclaration())
5      {
6          if( state.expression.getType() == null){
7              errors.add(new IllegalTypeException("Expected int but
                   received nothing ", state.getLocation()));
8
9          } else {
10             errors.add(new IllegalTypeException("Expected int but
                   received " + state.expression.getType().getSpelling
                   (), state.getLocation()));
11         }
12         return null;
13     }
14
15     table.openScope();
16     state.singleStatement.visit(this, arg);
17     table.closeScope();
18
19     if (state.singleStatement.returnsFully)
20
21     {
22         if (state.singleStatement.getReturnType()!=arg)
23         {
24             errors.add(new IllegalTypeException("Wrong return type
                   detected. Expected: " + arg.toString()));
25             return null;
26         }
27
28     }
29     state.setReturnType(state.singleStatement.getReturnType());
30
31     return null;
32 }
```

Code snippet 3.10: *An example of the implementation of the* `visitWhileStatement` *method in the visitor class* ***Checker***.

Chapter 4

Contextual Constraints

This chapter lists and explains the type and scope rules of Arongadongk along with the rules to ensure that a function returns. The type rules present the types along with a description of the rules that ensure a program is well-typed. The scope rules describe the life time, visibility, and bindings of variables as well as functions and function calls. These rules are then used for the contextual analysis to make sure the source code is valid.

4.1 Type Rules

The rules are generally kept formal, however, some rules are described informally to keep them simple. The rules are essential to defining which programs are valid in our language and to set guidelines for our checker.

The first section, Abstract Syntax, presents the abstract syntax which will be used throughout the Type Rules section. Section 4.1.2 shows the type environment and auxiliary function which is needed to make our type rules. In Arongadongk we want to have a type hierarchy to translate between types, e.g. integers to floats. This is described in section 4.1.3.

Some rules are excluded here because they are straight forward and should not need much explanation to understand. These are found in appendix A.

4.1.1 Abstract Syntax

In this section we will be using a simple syntax for our language. This abstract syntax does not differentiate between the different kinds of expressions – such as logical and comparison. The syntactic categories and their meta-variables for Arongadongk are:

$bl \in \mathbf{BLit}$ – Boolean Literals
$il \in \mathbf{IntLit}$ – Integer Literals
$fl \in \mathbf{FloatLit}$ – Float Literals
$sl \in \mathbf{StrLit}$ – String Literals
$lit \in \mathbf{Lit}$ – Literals
$t \in \mathbf{T}$ – Type Identifiers
$f \in \mathbf{Func}$ – Functions
$x \in \mathbf{Var}$ – Variables
$D \in \mathbf{Decl}$ – Declarations
$S \in \mathbf{Stm}$ – Statements
$MS \in \mathbf{MStm}$ – Multi-Statements
$apl \in \mathbf{APL}$ – Actual Parameter Lists
$fpl \in \mathbf{FPL}$ – Formal Parameter Lists
$papl \in \mathbf{PAPL}$ – Actual Parameter Lists
$pfpl \in \mathbf{PFPL}$ – Formal Parameter Lists
$e \in \mathbf{Exp}$ – Expressions

We will stick to the meta-variables given above when defining type rules. Should there be more than one presence of an element of the same category in the same rule we will be using indexes to differentiate between them.

Formation rules are the production rules of an abstract syntax. Our formation rules will be using BNF-style as used in section 3.1 and 3.2. The formation rules for our abstract syntax are as follows:

$$D ::= t\ f\ \mathbf{(}\ fpl\ \mathbf{)}\ S\ D_1 \mid \mathbf{const}\ t\ x = l\ \mathbf{;}\ D_1 \mid \mathbf{global}\ t\ x\ \mathbf{;}\ D_1 \mid \varepsilon \tag{4.1}$$

$$S ::= \mathbf{\{}\ MS\ \mathbf{\}} \mid \mathbf{if (}\ e\ \mathbf{)}\ S_1 \mid \mathbf{if (}\ e\ \mathbf{)}\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{while (}\ e\ \mathbf{)}\ S_1 \mid f\ \mathbf{(}\ apl\ \mathbf{)\ ;} \mid x = e\ \mathbf{;} \mid t\ x\ \mathbf{;} \mid \mathbf{return}\ e\ \mathbf{;} \tag{4.2}$$

$$MS ::= S\ MS_1 \mid \varepsilon \tag{4.3}$$

$$apl ::= e\ papl \mid \varepsilon \tag{4.4}$$

$$papl ::= \mathbf{,}\ e\ papl_1 \mid \varepsilon \tag{4.5}$$

$$fpl ::= t\ x\ pfpl \mid \varepsilon \tag{4.6}$$

$$pfpl ::= \mathbf{,}\ t\ x\ pfpl_1 \mid \varepsilon \tag{4.7}$$

$$
\begin{aligned}
e ::= & e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 \texttt{!=} e_2 \mid \\
& e_1 == e_2 \mid e_1 <= e_2 \mid e_1 >= e_2 \mid e_1 \mid\mid e_2 \mid e_1 \,\&\&\, e_2 \mid \\
& \texttt{!}e_1 \mid -e_1 \mid (e_1) \mid lit \mid x \mid f(apl)
\end{aligned} \tag{4.8}
$$

$$
lit ::= bl \mid il \mid fl \mid sl \tag{4.9}
$$

This abstract syntax is inspired by the one of Typed Bump [4, p. 187]. As formation rule 4.1 shows, we are using quite a different construction for declarations here compared to our concrete syntax in section 3.1.1. This syntax is simpler because we do not need a "Multiple Declaration" category.

As the formation rules for statements in section 4.2 shows, the *else-difference* is removed, and instead two different constructions are possible. Aside from that the abstract syntax for statements is similar to the concrete syntax of statements.

The abstract syntax for multi-statement is similar to the concrete syntax. The same applies for the parameters – both formal and actual.

The formation rules of the first two lines of formation rule 4.8 are called multiple expressions and the formation rules of the last line are called single expressions. The abstract syntax of expressions is different from the concrete with respect to the precedence order, because we only look at two operands and their type at the time. Aside from the fact that all expressions are generalized, it should be clear that it stems from the concrete syntax of expressions in section 3.1.1.4.

Formation rules for literals are not given as their constructions are the same as the literals given in our concrete syntax in section 3.1. The same applies for variables, functions, and type identifiers.

4.1.2 Environment & Auxiliary Function

We need a type environment to be able to define the type rules of Arongadongk. This environment and the auxiliary function are defined in this section.

We are using **Types** as the set of types, i.e. **Types** = {*Float*, *String*, *Int*, *Bool*}. We will be using τ as an element of **Types** throughout this section and it may be thought of as a meta-variable.

4.1.2.1 Type Environment

The type environment is used to transform a type identifier, t, into actual types and has the responsibility of binding variables, constants, and functions to types. We denote the type environment as E. E will initially be empty, i.e. there are no mappings. We define E as the following partial function:

$$E : \mathbf{T} \cup \mathbf{Var} \cup \mathbf{Func} \rightharpoonup \mathbf{Types}$$

We will use the notation $E\,[x \mapsto \tau]$ to express the following:

$$\text{let } E' = E\left[x \mapsto \tau\right] \text{ then}$$
$$E'\left(y\right) = \begin{cases} E\left(y\right) & \text{if } y \neq x \\ \tau & \text{if } y = x \end{cases}$$

[4, p. 190]

This is essential because it allows variables to be declared with a type. Notice that it is not only variables that can be bound to a type, but also functions and type identifiers. This is done in the exact same way.

4.1.2.2 Special Types

We allow the following to make the process of defining function declaration rules simple:

$$\tau ::= \tau_1 \rightarrow \tau_2 \tag{4.10}$$

Formation rule 4.10 means that the type τ_1 as input to a function will yield τ_2. It is used to bind functions, e.g. $E\left[f \mapsto Int \rightarrow Float\right]$ means that if f gets a variable of type Int as input it will return a *Float* [4, p. 190].

Since we allow multiple inputs for a function we allow cross product of types. This is done as follows: $\tau = \tau_1 \times \tau_2$. This means that we allow the following production for types:

$$\tau ::= \tau_1 \times \tau_2 \mid \varepsilon \tag{4.11}$$

The exact operation for making a cross product of types is not described here, but it follows the standards for products.

It is important that **Types** and **T** are not confused. Although closely related, they do not represent the same set. **T** is a set containing identifiers, namely **int**, **float**, **string**, and **bool**, while **Types** contains the actual types *Int*, *Float*, *String*, and *Bool*. Also notice that although the meta-variables are similar, they do not represent the same thing.

4.1.2.3 Auxiliary Function

We will be using an auxiliary function aux_E to allow changes in the environment to pass from one declaration to another and from one statement to another [4, p. 191]. aux_E is defined as follows:

[GLOBAL$_\text{E}$]
$$aux_E(\textbf{global } t\ x\ \textbf{;}\ D, E) = aux_E(D, E\left[x \mapsto E\left(t\right)\right]) \tag{4.12}$$

[CONST$_\text{E}$]
$$aux_E(\textbf{const } t\ x = lit\ \textbf{;}\ D, E) = aux_t(D, E\left[x \mapsto E\left(t\right)\right]) \tag{4.13}$$

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

$$aux_E(t\ f\ (fpl)\ S\ D, E) = aux_E(D, E\,[f \mapsto \tau \to E\,(t)]) \quad \text{where } aux_e\,(fpl, E) \vdash fpl : \tau \tag{4.14}$$

[EMPTY$_{\text{E}}$]

$$aux_E(\varepsilon, E) = E \tag{4.15}$$

[LOCAL-DECL$_{\text{E}}$]

$$aux_E(t\ x\ ;, E) = E\,[x \mapsto E\,(t)] \tag{4.16}$$

[OTHER-STM$_{\text{E}}$]

$$aux_E(\text{other } S \text{ formation rule}, E) = E \tag{4.17}$$

[FPL$_{\text{E}}$]

$$aux_E(t\ x\ pfpl, E) = aux_E\,(pfpl, E\,[x \mapsto E\,(t)]) \tag{4.18}$$

[PFPL$_{\text{E}}$]

$$aux_E(,t\ x\ pfpl, E) = aux_E\,(pfpl, E\,[x \mapsto E\,(t)]) \tag{4.19}$$

Rule 4.14 uses the rule for *fpl*, which is defined in section 4.1.6.

4.1.3 Type Hierarchy

Before presenting our actual type rules, we will define type hierarchy. We will use the expression $\tau_1 <: \tau_2$ to denote that τ_1 is a subtype of τ_2. Any value which is of type τ_1 is also of type τ_2.

We choose our integer type to be a subtype of our floating point type since any integer might be represented as a floating point number. We write this as the following:

[INT-FLOAT$_{\text{HIER}}$]

$$Int <: Float \tag{4.20}$$

We will be using the notation $\tau_1 \leq \tau_2$ to denote the following: $(\tau_1 <: \tau_2) \vee (\tau_1 = \tau_2)$.

We introduce a new type called VOID, which is a supertype to all other types. The programmer is, however, not able to use it. It is only used to indicate that two types are incompatible. Notice that VOID is a part of the **Types** set.

Beside VOID, we also add OK to the **Types**. We use it as the most specific subtype, which is a subtype of each type. That means that for any $\tau \in$ **Types** the following holds:

[OK-VOID$_{\text{HIER}}$]

$$\text{OK} \leq \tau \leq \text{VOID} \tag{4.21}$$

Like VOID, OK cannot be used by the programmer, it is merely an internal type.

In our type hierarchy the associative rule applies:

[ASSOC$_{\text{HIER}}$]

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \tag{4.22}$$

In our type rules we will be using the arrow notation, $\tau_1 \to \tau_2$, the hierarchy rules for this construction are:

[INPUT$_{\text{HIER}}$]

$$\frac{\tau_2 <: \tau_1}{\tau_1 \to \tau <: \tau_2 \to \tau} \tag{4.23}$$

[OUTPUT$_{\text{HIER}}$]

$$\frac{\tau_1 <: \tau_2}{\tau \to \tau_1 <: \tau \to \tau_2} \tag{4.24}$$

We also allow cross products of types. The hierarchical rule for such constructs is as follows:

[PRODUCT$_{\text{HIER}}$]

$$\frac{\tau_1 <: \tau_2}{\tau \times \tau_1 <: \tau \times \tau_2} \tag{4.25}$$

A complete diagram of the type hierarchy can be seen in figure 4.1.

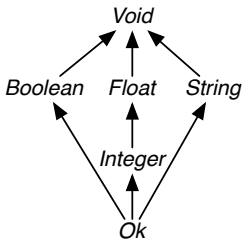


Figure 4.1: *The type hierarchy.*

4.1.4 Declarations

The type rule of the empty declaration is omitted but can be found in appendix A.2.

For a function call to be well-typed the type name must evaluate to a type, the function name should evaluate to OK, which means it is not previously declared. The single statement within must be well-typed and return the same type as indicated by the type name. The formal parameter list is saved by the auxiliary method in the type environment.

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

$$\frac{\begin{array}{rcl} E & \vdash & t : \tau \\ E' & \vdash & S : \tau \\ E'' & \vdash & D : \textsc{Ok} \end{array}}{E \vdash\ t\ f(fpl)\ S\ D : \textsc{Ok}} \tag{4.26}$$

$$\begin{array}{rl} \text{where} & E' = aux_E(fpl, E) \\ \text{and} & E' \vdash fpl : \tau' \\ \text{and} & E'' = E\,[f \mapsto \tau' \to \tau] \end{array}$$

The constant declaration rule must check that the type is valid and that the literal evaluates to the same type as stated by the type name.

[CONSTANT$_{\text{DECL}}$]

$$\frac{E \vdash t : \tau \quad E \vdash lit : \tau \quad E\,[x \mapsto \tau] \vdash D : \textsc{Ok}}{E \vdash \textbf{const}\ t\ x = lit\ \textbf{;}\ D : \textsc{Ok}} \tag{4.27}$$

4.1.5 Statements

Type rules for single-statements and multi-statements have the following formats:

$$E \vdash S\ : \tau \quad E \vdash MS\ : \tau \tag{4.28}$$

τ represents the return type of the given statement or multi-statement. Should a statement (or multi-statement) not return, but still be well-typed, the statement evaluates OK.

If two different types are returned, there will be found a common supertype for these two types according to our type hierarchy in section 4.1.3.

Notice that there is always at least one common supertype, VOID.

To allow a specific type to be lifted to a more general type we use the following rule for statements:

[SUBSUMPTION$_{\text{STM}}$]

$$\frac{E \vdash S : \tau_1 \quad \tau_1 <: \tau_2}{E \vdash S : \tau_2} \tag{4.29}$$

The exact same rule applies for multi-statement, it is found in appendix A.

The type rules for statements are presented below.

[BLOCK$_{\text{STM}}$]

$$\frac{E \vdash MS\ : \tau}{E \vdash \textbf{\{}MS\textbf{\}}\ : \tau} \tag{4.30}$$

[IF$_{\text{STM}}$]

$$\frac{\begin{array}{rcl} E & \vdash & e : Bool \\ E & \vdash & S : \tau \end{array}}{E \vdash \textbf{if}(e)\ S\ : \tau} \tag{4.31}$$

The if-else-statement has two inner statements, which must have the same type. Note that these statements may use the subsumption rule to lift the type of any or both of the two inner statements to a more general one.

[IF-ELSE$_{\text{STM}}$]

$$\frac{\begin{array}{l} E \vdash e : \mathit{Bool} \\ E \vdash S_1 : \tau \\ E \vdash S_2 : \tau \end{array}}{E \vdash \textbf{if(}e\textbf{)}\ S_1\ \textbf{else}\ S_2\ : \tau} \tag{4.32}$$

In type rule 4.32 the auxiliary method is not used. This is because they are using their own scopes.

[WHILE$_{\text{STM}}$]

$$\frac{E \vdash e : \mathit{Bool} \quad E \vdash S\ : \tau}{E \vdash \textbf{while(}e\textbf{)}\ S\ : \tau} \tag{4.33}$$

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

$$\frac{E \vdash \mathit{apl} : \tau}{E \vdash f\textbf{(}\ \mathit{apl}\ \textbf{)}\ \textbf{;} : \text{OK}} \tag{4.34}$$

$$\text{where} \quad E\,(f) = \tau \to \tau'$$

[ASSIGNMENT$_{\text{STM}}$]

$$\frac{E \vdash x : \tau \quad E \vdash e : \tau}{E \vdash x = e\ \textbf{;} : \text{OK}} \tag{4.35}$$

Expressions also have a subsumption rule to lift the type of e in type rule 4.35.

[LOCAL-DECLARATION$_{\text{STM}}$]

$$\frac{E \vdash t : \tau}{E \vdash t\ x\ \textbf{;} : \text{OK}} \tag{4.36}$$

Notice that we do not check whether x is used before. This is because our scope rules are too complicated to incorporate into our type system in this rule, instead see section 4.2 to see our scope rules.

[RETURN$_{\text{STM}}$]

$$\frac{E \vdash e : \tau}{E \vdash \textbf{return}\ e\ \textbf{;}\ : \tau} \tag{4.37}$$

[MULTI$_{\text{STM}}$]

$$\frac{E \vdash S : \tau \quad E' \vdash MS : \tau}{E \vdash S\ MS : \tau} \qquad (4.38)$$

$$\text{where} \quad E' = aux_E(S, E)$$

Notice that in type rule 4.38 the auxiliary function aux_E is used to update the type environment between statements.

[EMPTY$_{\text{STM}}$]

$$E \vdash \boldsymbol{\varepsilon} : \text{OK} \qquad (4.39)$$

4.1.6 Parameters

Every parameter list has an empty rule, which is equal to the following rule. The empty rule is used to end the proper parameter lists and in case a function has no parameters.

[EMPTY$_{\text{PARAM}}$]

$$E \vdash \boldsymbol{\varepsilon} : \varepsilon \qquad (4.40)$$

To allow a specific type of a parameter to be lifted to a more general type we use the following rule:

[SUBSUMPTION$_{\text{PARAM}}$]

$$\frac{E \vdash param : \tau_1 \quad \tau_1 <: \tau_2}{E \vdash param : \tau_2} \qquad (4.41)$$

$$\text{where} \quad param \in \mathbf{FPL} \cup \mathbf{PFPL} \cup \mathbf{APL} \cup \mathbf{PAPL}$$

The rule for the formal parameter list is used for the starting parameter of a parameter list. This rule can be an empty rule if the function takes no parameters. For a parameter list to be well-typed t must evaluate to a type and the proper formal parameter must be well-typed. The type of the formal parameter is a tuple of types. This tuple has the type of the variable x as the first element and the type tuple of the proper formal parameter list as the rest.

[ACTUAL$_{\text{FPL}}$]

$$\frac{E \vdash t : \tau_1 \quad E \vdash pfpl : \tau_2}{E \vdash t\ x\ pfpl : \tau_1 \times \tau_2} \qquad (4.42)$$

The actual parameter list rule is used for parameters when calling a function. An actual parameter list is well-typed if its expression evaluates to a type and the proper actual parameter list is also well-typed. As with the formal parameter list the type is made through cross product of the expression and the proper actual parameter list.

[ACTUAL$_{\text{APL}}$]

$$\frac{E \vdash e : \tau_1 \quad E \vdash \mathit{papl} : \tau_2}{E \vdash e\ \mathit{papl} : \tau_1 \times \tau_2} \tag{4.43}$$

The types are collected in a tuple, so it can be checked that the formal parameters and the actual parameters' types match up. This can be seen in the function call statement. Both actual and formal have a proper variant, which is in appendix A.3. The only difference between the proper rule and the non-proper rule is a leading comma.

4.1.7 Expressions

In this section the type rules for our expressions are defined.

To allow a specific type of an expression to be lifted to a more general type we use the following rule:

[SUBSUMPTION$_{\text{EXP}}$]

$$\frac{E \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{E \vdash e : \tau_2} \tag{4.44}$$

A binary expression is well-typed if the first and second expression evaluates to the same type (remember that types can be lifted with subsumption), and that this type is compatible with the operator O.

The unary expression type rule has been omitted since it is trivial. It can be found in appendix A.4.

[LOGICAL$_{\text{EXP}}$]

$$\frac{E \vdash e_1 : \tau \quad E \vdash e_2 : \tau \quad \tau = \mathit{Bool}}{E \vdash e_1\ O\ e_2 : \mathit{Bool}} \tag{4.45}$$

$$\text{where} \qquad O \in \{||, \&\&\}$$

[EQUALITY$_{\text{EXP}}$]

$$\frac{E \vdash e_1 : \tau \quad E \vdash e_2 : \tau}{E \vdash e_1\ O\ e_2\ : \mathit{Bool}} \tag{4.46}$$

$$\text{where} \qquad O \in \{\mathbf{!}=, ==\}$$

[COMPARISON$_{\text{EXP}}$]

$$\frac{E \vdash e_1 : \tau \quad E \vdash e_2 : \tau \quad \tau = \mathit{Float}}{E \vdash e_1\ O\ e_2\ : \mathit{Bool}} \tag{4.47}$$

$$\text{where} \qquad O \in \{<, >, <=, >=\}$$

[ADD$_{\text{EXP}}$]

$$\frac{E \vdash e_1 : \tau \quad E \vdash e_2 : \tau}{E \vdash e_1 + e_2\ : \tau} \tag{4.48}$$

$$\text{if} \quad \tau = \mathit{String} \vee \mathit{Float}$$

[ARITHMETIC$_{EXP}$]

$$\frac{E \vdash e_1 : \tau \quad E \vdash e_2 : \tau \quad \tau = Float}{E \vdash e_1\ O\ e_2\ : \tau} \tag{4.49}$$

$$\text{where} \quad O \in \{-, *, /\}$$

Notice that if a type τ is a subtype of type *Int* it is automatically a subtype of *Float*. This means that operands of both type *Float* and *Int* are allowed in type rules 4.47, 4.48, and 4.49.

The type rules for unary expressions are straight forward and are not shown here. These are found in appendix A.4.

4.1.7.1 Factor

For a function call to be well-typed the function's actual parameter list needs to be of the same type as the function's formal parameter list.

[CALL$_{EXP}$]

$$\frac{E \vdash apl : \tau_1}{E \vdash f(\ apl\ ) : \tau_2} \tag{4.50}$$

$$\text{where} \quad E(f) = \tau_1 \rightarrow \tau_2$$

[NESTED$_{EXP}$]

$$\frac{E \vdash e : \tau}{E \vdash (e) : \tau} \tag{4.51}$$

[VAR$_{EXP}$]

$$E \vdash x : \tau \tag{4.52}$$

$$\text{where} \qquad \tau = E\,(x)$$

4.1.7.2 Literal

The following type rules apply for the Arongadongk literals:

[BOOLEAN$_{EXP}$]

$$E \vdash bl : Bool \tag{4.53}$$

[INT$_{EXP}$]

$$E \vdash il : Int \tag{4.54}$$

[FLOAT$_{EXP}$]

$$E \vdash fl : Float \tag{4.55}$$

[STRING$_{EXP}$]

$$E \vdash sl : String \tag{4.56}$$

4.2 Scope Rules

This section describes the scope rules for our language. The scope rules include lifetime, visibility, and bindings of variables. The scope rules also include bindings of functions. How the semantics of the scope rules work is described in section 5.1.

For functions Arongadongk exhibits monolithic block structure [12, p. 137-139], which means all functions are visible throughout the whole program and cannot be overridden or hidden.

In Arongadongk constants, local variables, and global variables the scope rules exhibit nested block structure [12, p. 142-148]. This implies that several scope levels are needed and that variables have a lifetime, which is not necessarily as long as that of the program.

4.2.1 Opening and Closing Scopes

In Arongadongk a scope opens when a left curly bracket ("{") occurs and closes again when a right curly bracket ("}") occurs. Furthermore we open scopes when we enter the bodies of if-statements and while-statements, and we close scopes when we leave. This is because we allow the single-statement, which is the body of an if-statement to be a declaration-statement. Should a new variable be declared in such a way, it will not be visible outside of the if-statement. The same holds for the while-statement and the else-statement.

A function declaration opens a scope for its formal parameters, which will be closed at the end of the function declaration, because these are not supposed to be visible in the global scope. If the body of a function is a block statement, as we state to be good practice in section 3.1.1.2, yet another scope will be opened.

4.2.2 Scope Levels

A statement or an expression in scope x can access any variable declared in scope y as long as $y \leq x$. The same applies for functions and constants, but these can only be declared in scope level 1 (or 0 if they are part of the standard environment).

Whenever a scope is closed all variables declared inside that scope will seize to exist. The scope level where global variables, constants, and functions are declared is called the global scope and is given the scope level 1. Each time a scope is opened, the scope level is incremented. Each time a scope is closed, the scope level is decremented. A scope cannot be closed before one has been opened. This means that a programmer using Arongadongk can never write anything in scope level 0.

4.2.3 Hiding of Variables

A statement-declaration in scope level x can declare a variable which has the same name as a variable in scope level y, where $y < x$. This will hide the variable in the outer scope. It will however become visible again when going out of scope level x.

This means that the same variable name could be used several times as long as it is within a new scope each time. This grants the user the opportunity to

use variable names freely without running into name collisions, and opens the possibility of confusing the programmer.

4.2.4 Function Calls

Function calls in Arongadongk exhibits static scope rules for variables. This means that any variable used in a body of a function must refer to either a global variable or a variable declared locally inside the given function. Constants can also be accessed in the body of a function. Code snippet 4.1 illustrates source code, where the scope rules for function calls matters. In code snippet 4.1 the variable i in lines 9 and 12 both refer to the local declaration in line 7. On the other hand the variable i in line 3 refers to the global integer declaration in line 1, although `foo` is called by `bar` in line 11.

```
1  global int i;
2  int foo() {
3      i = i + 1;
4      return 0;
5  }
6  int bar() {
7      int i;
8      i = 0;
9      while(i < 10)
10     {
11         foo();
12         i = i + 1;
13     }
14     return 0;
15 }
```

Code snippet 4.1: *Illustration of source code where the same variable name refers to different declarations at different points within it.*

The binding of function declarations upon function calls does not matter because we only allow functions to be declared in the global scope.

4.3 Ensure Return Value of Functions

We want the functions of our language to return at some point. In section 4.1 we ensure that it is the correct type, which is returned. In this section we will ensure that a function actually returns at some point. If the body of a function simply ends without using a return statement then we have to decide what to send back to the call site.

One option could be that we choose some standard value to be returned if no return statement is encountered. Instead we prefer to make a static check of the source code to ensure that each function actually will return a value.

Furthermore, we do not allow code after a point where a return is inevitable. The reason for this is that it should be easier to implement when translating to Java, where such “unreachable code” is not allowed.

The rules we present are similar to the type rules for statements, seen in section 4.1.5, except that they state whether a function returns, and not which type the function returns.

The rules are on the form:

$$\begin{array}{ll} S : r & MS : r \\ \text{where} & r \in \{tt, ff, error\} \end{array} \tag{4.57}$$

The r symbol represents the value, which indicates whether or not a statement or multi-statement is ensured to return. The *error* is used whenever a statement occurs after a return has been ensured, i.e. when unreachable code is detected. Notice that we are using the same abstract syntax as we did with type rules. The formation rules for these are found in section 4.1.1.

The return rule for block statements is simple. It ensures that they return correctly if the internal multi-statement returns correctly. This is seen in return rule 4.58.

[BLOCK$_{\text{STM}}$]

$$\frac{MS : r}{\{MS\} : r} \tag{4.58}$$

An if-statement (without an else clause) is not guaranteed to return, since the body of the if-statement may not run.

[IF$_{\text{STM}}$]

$$\frac{S : r}{\textbf{if}(e)\ S : ff} \quad \text{if} \quad r \neq error \tag{4.59}$$

[IF-ERROR$_{\text{STM}}$]

$$\frac{S : error}{\textbf{if}(e)\ S : error} \tag{4.60}$$

The if-else-statement has two inner statements, which must both ensure return for the entire if-else-statement to ensure return.

[IF-ELSE$_{\text{STM}}$]

$$\frac{S_1 : r_1 \quad S_2 : r_2}{\textbf{if}(e)\ S_1\ \textbf{else}\ S_2 : r} \tag{4.61}$$

$$\begin{array}{rl} \text{if} & (r_1 \neq error) \wedge (r_2 \neq error) \\ \text{where} & r = r_1 \wedge r_2 \end{array} \tag{4.62}$$

[IF-ELSE-ERROR$_{\text{STM}}$]

$$\frac{S_1 : r_1 \quad S_2 : r_2}{\textbf{if}(e)\ S_1\ \textbf{else}\ S_2 : error} \tag{4.63}$$

$$\text{if} \quad (r_1 = error) \vee (r_2 = error)$$

We allow ourselves to use logical and ($\wedge$) between r_1 and r_2 in return rule 4.61, since we have checked that both are boolean values, because none of them are *error*.

A while-statement exhibits the same behavior as an if-statement and can be found in appendix B.

Function-call-statement, assignment-statement, and local-declaration-statement can never return and will always yield *ff*. The exact rules for these three statements can be found in appendix B.

A return statement will yield *tt* since this will ensure that a return occurs.

[RETURN$_{\text{STM}}$]

$$\textbf{return}\ e\ \textbf{;} : tt \tag{4.64}$$

A multi-statement is said to ensure return if and only if the last statement within it ensures return. Should an earlier statement ensure return an *error* is reported. If no statement within a multi-statement ensures return, then neither does the multi-statement. An empty multi-statement cannot ensure return.

[MULTI$_{\text{STM}}$]

$$\frac{S : ff \quad MS : r}{S\ MS : r} \tag{4.65}$$

[MULTI-RETURN$_{\text{STM}}$]

$$\frac{S : tt}{S\ MS : tt} \quad \text{if} \quad MS = \varepsilon \tag{4.66}$$

[MULTI-ERROR-1$_{\text{STM}}$]

$$\frac{S : error}{S\ MS : error} \tag{4.67}$$

[MULTI-ERROR-2$_{\text{STM}}$]

$$\frac{S : tt}{S\ MS : error} \quad \text{if} \quad MS \neq \varepsilon \tag{4.68}$$

[EMPTY$_{\text{STM}}$]

$$\varepsilon : ff \tag{4.69}$$

4.4 Contextual Analysis

The following sections outline how the contextual analysis is implemented in our compiler. The goal is to verify that the contextual constraints specified by the Arongadongk language – meaning the type rules in section 4.1, the scope rules in section 4.2, and the return rules in section 4.3 – are not violated.

4.4.1 Checker

The **Checker** class has a `check` method, which takes the AST as an argument and traverses it while reporting any contextual errors discovered. The **Checker** class uses the visitor pattern described in 3.5.4. The **Checker** class decorates the AST, with the declarations of the variables, types, functions, and constants. After this, it is said to be decorated.

4.4.1.1 Identification Table

While traversing the AST, an identification table is used to store declarations such as variable declarations (including constants and global variables), function declarations, and type declarations. The purpose of the identification table is to verify that calls to declarations are valid. An example is an assignment of a variable, which has not yet been declared. See rules 4.35 and 5.14. Note that although user defined types are not part of the Arongadongk language, the identification table still supports type declarations to make it easier for us to implement a standard environment. The identification table also handles scope levels. This is done by the `openScope` and `closeScope` methods, illustrated in code snippet 4.2. The `table` object is an instance of the identification table.

```
1  public Object visitFunctionDeclaration(FunctionDeclaration decl,
       Object arg) {
2
3      table.openScope();
4
5      decl.parameterList.visit(this, decl);
6      decl.singleStatement.visit(this,decl.getType());
7
8      if (!decl.singleStatement.returnsFully) {
9          errors.add(new MissingReturnException("Function does not
               return properly", decl));
10     }
11
12     table.closeScope();
13
14     return null;
15 }
```

Code snippet 4.2: *The `visitFunctionDeclaration` method, illustrating usage of `openScope` and `closeScope`.*

Since the Arongadongk language specifies that it is not possible to access declarations declared in a scope, which is more deeply nested, it makes no sense to keep declarations in the identification table whenever scopes are closed. Therefore all declarations within a certain scope are deleted from the identification table whenever that scope is closed. This is illustrated in the `closeScope` method in code snippet 4.3. Note that `currentLevel` is used whenever inserting a new declaration into the table. See section 4.2 for scope rules.

The identification table has been implemented as a list of so called "Tuples", which are objects of the class **Tuple**. The only purpose of a Tuple is to hold the identifier, the declaration, and the current scope level.

4.4.1.2 Coercer

The way we have implemented the type hierarchy as explained in section 4.1.3 is via a coercer. In Arongadongk there are three different scenarios where the types can change according to the type hierarchy: assignments, binary operations, and unary operations. The **Coercer** deals with these scenarios so the types stay according to the type hierarchy. Whenever visiting nodes on the AST which are of a kind that needs coercion, the **Checker** utilizes the **Coercer** to coerce the types. An example is shown in code snippet 4.4.

```
1  public class IdentificationTable implements IIdentificationTable
2  {
3      public ArrayList<Tuple> list;
4      private int currentLevel = 0;
5
6      public  IdentificationTable(){
7          list = new ArrayList<Tuple>();
8      }
9      ...
10     public void openScope() {
11         currentLevel++;
12     }
13
14     public void closeScope() {
15         Iterator<Tuple> i = list.iterator();
16         while(i.hasNext()) {
17             Tuple t = i.next();
18             if(t.level == currentLevel) {
19                 i.remove();
20             }
21         }
22         currentLevel--;
23     }
24     ...
25 }
```

Code snippet 4.3: *Part of the **Identification Table** class. Code has been omitted for readability.*

The variable **arg** is a **TypeDeclaration**. The value of **arg** is the expected type of **returnExpression**.

Knowing what the function expects to return, the `visitReturnStatement` can safely utilize the coercer to check if it should report an error or not.

Notice in code snippet 4.4 that the **returnsFully** variable is only set to true if the type is correct. The **returnsFully** variable is our implementation of the return rules in the section 4.3. The *error* state (in section 4.3) is only found in multi-statement where an error is added to the error list of the **Checker** if the multi-statement returns before the last statement has been encountered. This can be seen in code snippet 4.5. Note that in our implementation of the abstract syntax tree the multi-statement is the left part of the multi-statement, which is opposite of our grammar.

4.4.1.3 Declaration Checker

Code snippet 4.6 illustrates a typical example of "ping-pong" function calls. It would not help to change the order of the declarations because both functions calls each other.

We have chosen to make an additional pass in the beginning of the contextual analysis and it is handled by the **DeclarationChecker** class. Since the Arongadongk CFG specifies that functions cannot be declared inside other functions, the Declaration Checker can safely stop traversing a single-statement body that any function declaration has.

```
1  public Object visitReturnStatement(ReturnStatement state, Object
       arg) {
2
3      state.returnExpression.visit(this, arg);
4
5      if (Coercer.Assignment(state.returnExpression.getType() ,(
           TypeDeclaration)arg) == null) {
6          errors.add(new IllegalTypeException("return statement:
               Wrong return type", (TypeDeclaration)arg, state.
               returnExpression.getType(), state.getLocation()));
7          state.returnsFully = false;
8      } else {
9          state.returnsFully = true;
10     }
11     state.setReturnType(Coercer.Assignment(state.returnExpression.
           getType() ,(TypeDeclaration)arg));
12
13     return null;
14 }
```

Code snippet 4.4: *The `visitReturnStatement` method.*

```
1  public Object visitMultipleStatement(MultipleStatement state,
       Object arg) {
2      state.returnsFully = false;
3
4      state.statement.visit(this, arg);
5      state.singleStatement.visit(this, arg);
6
7      ...
8
9      if (state.statement.returnsFully) {
10         state.returnsFully = true;
11         errors.add(new CheckerException("Unreachable code detected
               at " + state.singleStatement.getLocation()));
12     } else if (state.singleStatement.returnsFully) {
13         state.returnsFully = true;
14     }
15     return null;
16 }
```

Code snippet 4.5: *The `visitMultipleStatement` method in the **Checker** class. Code which checks and sets returntype has been omitted.*

```
1  int foo() {
2      if(...)
3          return bar(0);
4      return bar(5);
5  }
6  int bar(int baz) {
7      while(...)
8          baz = foo() + baz;
9      return baz;
10 }
```

Code snippet 4.6: *Example of "ping-pong" between functions `foo` and `bar`.*

Chapter 5

Semantics

This chapter presents the formal description of the semantic rules of Arongadongk. Apart from the semantics described by the semantic rules, there are some built-in functionalities which allow a developer to easily construct a graphical application. All this is used by the code generator to create an application based on the decorated abstract syntax tree, thereby completing the last phase of the compiler.

5.1 Semantic Rules

The semantic rules used for Arongadongk are inspired by the semantic rules of the language Bump [4, chapter 7]. We are primarily using formal rules to describe the semantics of our language. However, on some occasions we will use an informal description to avoid complex rules. Only non-trivial rules are presented in the report. The rest is included in appendix C. For the semantics we use the abstract syntax defined in section 4.1.1.

We have chosen to make a big-step-semantic, since we already decided to make a sequential language, which suits a big-step-semantic well. If we had chosen to focus on parallelism a small-step-semantic would have been preferred, since making parallelism in a big-step-semantic is impossible [4, p. 77]. As we care about neither parallelism nor non-determinism [4, p. 75], we are free to choose which operational semantic we prefer.

5.1.1 Environments

In the semantics rules of Arongadongk several environments are needed to store information. Some of these environment are based on the environment-store model presented in Transitions and Trees [4, p. 80].

5.1.1.1 Variable Environments & Stores

To model variables and bindings we introduce the environment-store model. It consist of an variable environment and a store. The environment maps variables to locations and the store maps locations to values. See figure 5.1. The locations are natural numbers [4, p. 81] and **Loc** is the set of all locations. **Values** is the set of all possible values in Arongadongk.

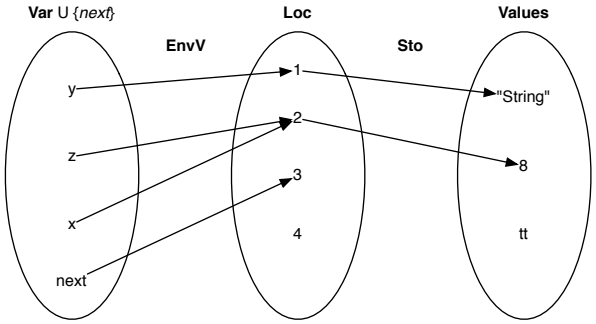


Figure 5.1: *The environment-store model.*

We introduce a *new* function, which gives the next location. We also introduce a pointer called *next*, which is bound to the next available location.

$$new : \mathbf{Loc} \to \mathbf{Loc} \tag{5.1}$$

More formally the set of variable environments is the set of partial functions from the set of variables and the *next* pointer to the set of locations:

$$\mathbf{EnvV} = \mathbf{Var} \cup \{next\} \rightharpoonup \mathbf{Loc} \tag{5.2}$$

We write env_v as an arbitrary element of **EnvV**.

The set of stores is a set of partial functions from locations to values [4, p. 81].

$$\mathbf{Sto} = \mathbf{Loc} \rightharpoonup \mathbf{Values} \tag{5.3}$$

sto denotes an arbitrary element of **Sto**

For an arbitrary element of any of these environments E we use the following notation:

$$E' = E[x \mapsto v] \tag{5.4}$$

The meaning of it is the same as described in section 4.1.2.1 e.g. the store is updated with the following notation:

$$sto' = sto[l \mapsto v] \text{where } l \text{ is a location and } v \text{ is a value}$$

5.1.1.2 Function Environments

A function environment maps function names to statements, formal parameter lists, and variable environments. The set of function environments is formally defined as:

$$\mathbf{EnvF} = \mathbf{Func} \rightharpoonup \mathbf{Stm} \times \mathbf{FPL} \times \mathbf{EnvV} \tag{5.5}$$

env_f denotes an arbitrary element in **EnvF**.

5.1.1.3 Return Environment

To save the value of a function we use a set of return environments called **Ret**. A return environment has two possible inputs, *return* and *result*. The *return* flag is used to determine that a function has returned, which means that execution of

the remaining body can be halted. The return value is saved in the environment as *result*. The return environment is defined as:

$$\mathbf{Ret} = \{result\} \cup \{return\} \rightharpoonup \mathbf{Values} \tag{5.6}$$

We write *ret* to denote an arbitrary element in **Ret**.

5.1.2 Program

The program rule starts by processing the declaration, D, then executes the setup function, and if the setup function evaluates to "true" the action function is executed in a while-loop terminating when action returns "false".

The semantics of the statement in the premise are explained in section 5.1.4.

[ACTUAL$_{\text{PROG}}$]

$$\frac{\langle D, env_f, env_v, sto\rangle \underset{decl}{\longrightarrow} (env'_f, env'_v, sto') \quad env'_f \vdash \langle \textbf{if (setup ( )) while (action ( )) \{ }\textbf{}, env'_v, sto', ret\rangle \underset{stm}{\longrightarrow} (env''_v, sto'', ret'')}{\langle D, env_f, env_v, sto\rangle \underset{prog}{\longrightarrow} (env''_f, env''_v, sto'')} \tag{5.7}$$

$$\begin{aligned} \text{where} \quad & env_f = \text{new std } env_f \\ \text{and} \quad & env_v = \text{new std } env_v \\ \text{and} \quad & sto = \text{new std } sto \\ \text{and} \quad & ret_e = \text{new empty } ret \\ \text{and} \quad & ret = ret_e[return \mapsto ff] \end{aligned}$$

As semantic rule 5.7 shows we are processing each declaration before starting the program to allow the "ping-pong" of functions. This is implemented in the checker and described in section 4.4.1.3.

Notice that the standard environments (*std*) are initialized in semantic rule 5.7. The standard environments consists of variables, values, and functions.

5.1.3 Declarations

A function declaration, as seen in semantic rule 5.8, saves the following in the function environment: The body of the function, which is a single statement, the formal parameter list, and the variable environment. It does not change the variable environment. It only saves the variables from the formal parameter list in a temporary variable environment, which are saved in the function environment.

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

$$\frac{\langle fpl, env_v\rangle \underset{fpl}{\longrightarrow} env'_v \quad \left\langle D, env'_f, env_v, sto\right\rangle \underset{fpl}{\longrightarrow} (env''_f, env''_v, sto')}{\langle t\, f\, (\ fpl\ )\ S\ D, env_f, env_v, sto\rangle \underset{decl}{\longrightarrow} (env''_f, env''_v, sto')} \tag{5.8}$$

$$\text{where } env'_f = env_f[f \mapsto S, fpl, env'_v]$$

The constants in Arongadongk cannot be changed, though this cannot be derived from the semantics. The store returned from the expression rule evaluating

the literal is not used, since a literal itself cannot change the store. This is seen for integer literals in rule 5.23.

[CONSTANT$_{\text{DECL}}$]

$$\frac{\langle D, env_f, env'_v, sto' \rangle \underset{decl}{\longrightarrow} (env'_f, env''_v, sto'')}{\langle t\ x = lit\ ;\ D, env_f, env_v, sto \rangle \underset{decl}{\longrightarrow} (env'_f, env''_v, sto'')} \tag{5.9}$$

$$\begin{array}{lll} \text{where} & l = env_v(next) \\ \text{and} & env_f, env_v \vdash \langle lit, sto \rangle \underset{exp}{\longrightarrow} (v, sto') \\ \text{and} & env'_v = env_v\,[x \mapsto next]\,[next \mapsto new(l)] \\ \text{and} & sto' = sto\,[l \mapsto v] \end{array}$$

The semantic rule for global variable declarations is similar to that of constant declaration except for the fact that global variables are not initialized. The semantic rule for global variable is found in appendix C.1

5.1.4 Statements

A multi-statement in Arongadongk is either a list of statements or an empty multi-statement. The rule for an empty statement is found in appendix C.3. We have two rules for multi-statements: One for when it should continue executing and one where it should stop executing. It should stop executing if a return statement has been executed.

[CONTINUE$_{\text{MSTM}}$]

$$\frac{\begin{array}{c} env_f \vdash \langle S, env_v, sto, ret \rangle \underset{stm}{\longrightarrow} (env''_v, sto'', ret'') \\ env_f \vdash \langle MS, env''_v, sto'', ret'' \rangle \underset{stm}{\longrightarrow} (env'_v, sto', ret') \end{array}}{env_f \vdash \langle S\ MS, env_v, sto, ret \rangle \underset{stm}{\longrightarrow} (env'_v, sto', ret')} \tag{5.10}$$

$$\text{if} \qquad ret(return) = ff$$

[RETURN$_{\text{MSTM}}$]

$$env_f \vdash \langle S\ MS, env_v, sto, ret \rangle \underset{stm}{\longrightarrow} (env_v, sto, ret) \tag{5.11}$$

$$\text{if} \qquad ret(return) = tt$$

A block-statement cannot declare variables, which are visible outside the block-statement hence the variable environment is not changed. The store environment is changed, because variables for surrounding scopes might change value inside the block statement.

[BLOCK$_{\text{STM}}$]

$$\frac{env_f\ \langle MS, env_v, sto, ret \rangle \underset{stm}{\longrightarrow} (env'_v, sto', ret')}{env_f \vdash \langle \{\ MS\ \}, env_v, sto, ret \rangle \underset{stm}{\longrightarrow} (env_v, sto', ret')} \tag{5.12}$$

As for block-statements, if-statements opens a new scope, executes the first statement or the second statement, and then closes the scope. Therefore, the

variable environment is not changed. Should an if-statement's expression evaluate to "false", the second statement is run instead of the first statement. The exact rule is found in appendix C.3. There is a version of the if statement with only one statement. This can be found in appendix C.3.

[IF-ELSE-TRUE$_{\text{STM}}$]

$$\frac{env_f \vdash \langle S_1, env_v, sto'', ret\rangle \underset{stm}{\longrightarrow} (env_v', sto', ret')}{env_f \vdash \langle \textbf{if (}\ e\ \textbf{)}\ S_1\ \textbf{else}\ S_2, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v, sto', ret')} \quad (5.13)$$

$$\text{if} \qquad env_f, env_v \vdash \langle e, sto\rangle \underset{exp}{\longrightarrow} (tt, sto'')$$

The semantics of a while-statement is quite similar to that of an if-statement when the expression evaluates to "true", except that it runs itself again after running the statement that is its body. The rule for while-statement is located in appendix C.3.

The declaration-statement makes a new location in the variable environment. The variable name must be unique in the scope we are within. If we are in a higher scope the lower scope variable is hidden. We chose this in order to increase writability so new variable names are not needed for each nested scope, thus ignoring the possibility of confusion.

[DECLARATION$_{\text{STM}}$]

$$env_f \vdash \langle t\ x\ \textbf{;}, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v', sto, ret) \quad (5.14)$$

$$\begin{aligned} \text{where} \quad & env_v(next) = l \\ \text{and} \quad & env_v' = env_v[x \mapsto l][next \mapsto new(l)] \end{aligned}$$

The function-call-statement starts by retrieving the formal parameter list, the variable environment (from the declaration point), and the statement from the function. It then executes the actual parameter list, which updates the variable environment with the values from the actual parameter list. The updated variable environment is then used when executing the statement.

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

$$\frac{\begin{array}{c} env_f, env_v', env_v \vdash \langle apl, fpl, sto\rangle \underset{apl}{\longrightarrow} sto'' \\ env_f \vdash \langle S, env_v'[next \mapsto new(l)], sto'', ret\rangle \underset{stm}{\longrightarrow} (env_v'', sto', ret') \end{array}}{env_f \vdash \langle f\ \textbf{(}apl\textbf{)}\ \textbf{;}, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v, sto', ret)} \quad (5.15)$$

$$\begin{aligned} \text{where} \quad & env_f(f) = (S, fpl,\ env_v') \\ \text{and} \quad & l = env_v(next) \end{aligned}$$

Notice that the old return environment is passed ahead instead of the new ret'. This is because ret' maps the *return* variable to "true" and it is not intended that a function should return whenever it makes a function call.

The return-statement updates the return environment with the value of the expression and it updates the *return* variable in the return environment to "true". This way it can be determined when to stop executing the rest of the function body.

[RETURN$_{\text{STM}}$]

$$\frac{env_f, env_v, ret \vdash \langle e, sto\rangle \underset{exp}{\longrightarrow} (v, sto')}{env_f \vdash \langle \textbf{return } e \textbf{ ;}, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v, sto', ret')} \quad (5.16)$$

$$where \qquad ret' = ret[return \mapsto tt][result \mapsto v]$$

5.1.5 Parameters

The formal parameters make a new location in the variable environment for each variable used. These will be used when the function is called.

[ACTUAL$_{\text{FPL}}$]

$$\frac{\langle pfpl, env_v[x \mapsto next][next \mapsto new(l)]\rangle \underset{pfpl}{\longrightarrow} env'_v}{\langle t\ f\ pfpl, env_v, \rangle \underset{fpl}{\longrightarrow} env'_v} \quad (5.17)$$

$$\text{where } l = env_v(next)$$

The actual parameters use both the formal and actual parameters. The semantic rule 5.18 updates the variables from the formal parameters with the values of the actual parameters. It uses two different variable environments. One from the function declaration, which contains the variable names and one from the call site, which contains the values. In this way it can be checked that the formal and actual parameters match up and assign the values of the actual parameters to the variables of the formal parameters.

[ACTUAL$_{\text{APL}}$]

$$\frac{env_f, env_{vf}, env_{va} \vdash \langle papl, pfpl, sto''[l \mapsto v]\rangle \underset{papl}{\longrightarrow} sto'}{env_f, env_{vf}, env_{va} \vdash \langle e\ papl, t\ x\ pfpl, sto\rangle \underset{apl}{\longrightarrow} sto'} \quad (5.18)$$

$$\text{where} \quad env_f, env_{va}, sto \vdash e \underset{exp}{\longrightarrow} (v, sto'')$$
$$\text{and} \quad l = env_{vf}(x)$$

Both actual and formal parameters can be empty. The rule for this is found in appendix C.2.

Actual parameter lists and formal parameter lists both have a "proper rule". These rules are found in appendix C.2.

5.1.6 Expression

The semantic rules for expressions are described here. Below in semantic rule 5.19 the semantic rule for a plus expression is presented. Notice that to allow the semantics of this to be run the type rules of section 4.1 must be obeyed.

[PLUS$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 + e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{5.19}$$

$$\text{where} \quad v = v_1 + v_2$$

The rest of the rules for multiple expressions are very similar to that of plus, which is why they are omitted here and can be found in appendix C.4. One thing to notice is that we do not allow division by zero, which means that division does stand somewhat out from the other multiple expressions.

The semantic rules of the single expression, which are shown in the last line of production rules of abstract syntax 4.8. Some rules are however omitted here for brevity, but can be found in appendix C.4.

The semantics of logical negation production rule is seen in semantic rule 5.20. The semantic rule for arithmetic negation is quite similar to that of logical negation. The exact rule for arithmetic negation is found in appendix C.4.

[LOGICAL-NEGATION$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto')}{env_f, env_v \vdash \langle \mathbf{!}e_1, sto\rangle \underset{exp}{\longrightarrow} (v, sto')} \tag{5.20}$$

$$\text{where} \quad v = \neg v_1$$

The semantics of nested expressions are shown in semantic rule 5.21. The variable environment and store of the internal expression is simply transferred to the outer expression.

[NESTED$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v, sto')}{env_f, env_v \vdash \langle \mathbf{(}e_1\mathbf{)}, sto\rangle \underset{exp}{\longrightarrow} (v, sto')} \tag{5.21}$$

To evaluate boolean literals to "true" the semantic rule 5.22 is used. The false rule is very similar, and is therefore placed in the appendix, namely appendix C.4.

[BOOLEAN-TRUE$_{\text{EXP}}$]

$$env_f, env_v \vdash \langle bl, sto\rangle \underset{exp}{\longrightarrow} (tt, sto) \qquad \text{if} \qquad bl = \mathbf{true} \tag{5.22}$$

To evaluate integer literals we are using an auxiliary function called $\mathcal{N}$ [4, p. 32]. This function maps the integer literals to an actual number. Remember that a literal is just a string of characters and has no arithmetic value.

The function $\mathcal{N}$ simply takes the integer literal and converts it into an integer value, e.g. $\mathcal{N}(\mathbf{5}) = 5$. Semantic rule 5.23 shows how $\mathcal{N}$ is used in the semantics for integer literal.

Float literals and string literals have similar functions called $\mathcal{F}$ and $\mathcal{S}$ respectively. The rules for these literals are found in appendix C.4.

[INTEGER$_{\text{EXP}}$]

$$env_f, env_v \vdash \langle il, sto \rangle \underset{exp}{\longrightarrow} (v, sto) \qquad \text{where} \qquad v = \mathcal{N}\,(il) \tag{5.23}$$

The value of a variable is retrieved as described in semantic rule 5.24. As the rule shows the location associated with the variable is retrieved and the value at this location is the one, which the given variable evaluates to.

[VARIABLE$_{\text{EXP}}$]

$$env_f, env_v \vdash \langle x, sto \rangle \underset{exp}{\longrightarrow} (v, sto) \qquad \begin{array}{rl} \text{if} & l = env_v\,(x) \\ \text{and} & v = sto\,(l) \end{array} \tag{5.24}$$

A function call inside an expression might change the store because a function in Arongadongk can change the value of global variables. Because of this, the store environment – which is evaluated from a function – must be saved and become a part of the evaluation of the outer expression. The semantics of a function call within an expression is described in semantic rule 5.25.

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

$$\frac{\begin{array}{c} env_f, env'_v, env_v \vdash \langle apl, fpl, sto \rangle \underset{apl}{\longrightarrow} sto'' \\ env_f \vdash \langle S, env_v[next \mapsto new(l)], sto'', ret \rangle \underset{stm}{\longrightarrow} (env'_v, sto', ret') \end{array}}{env_f, env_v \vdash \langle f\,(\ apl\ ), sto \rangle \underset{exp}{\longrightarrow} (v, sto')} \tag{5.25}$$

$$\begin{array}{rl} \text{where} & env_f(f) = (S, fpl, env'_v) \\ \text{and} & l = env_v(next) \\ \text{and} & ret_e = \text{new empty } ret \\ \text{and} & ret = ret_e\,[return \mapsto ff] \\ \text{and} & v = ret'\,(result) \end{array}$$

5.2 Functionality & API

When creating a language with a specific purpose, the aim should be to add something that makes the language better than other languages for that specific purpose. The following sections explain the functionality and the API, which are implemented in Arongadongk.

5.2.1 Functionality

Since Arongadongk is a programming language designed to be used for the development of multi agent systems, it has some functionality that eases such a development. Section 1.2 explains the concepts of a multi agent system. Such concepts should be supported by a language that is specifically designed for creating multi agent systems.

There are two functions that must be present in any Arongadongk source code. These functions are `setup` and `action`, and they both return a boolean.

The function `setup` is the first function to be called, and it is only called once. The intention with the `setup` function is to set or initialize variables,

and generally set up the structure of the program. If `setup` returns "false" the program will terminate, but if it returns "true" the `action` function is called.

The `action` function is continuously called by a loop, which runs as long as `action` returns "true". Between each call of the function `action` there is a delay, which can be set by an API call (see section 5.2.2). If the delay is one second it will run its body and then wait a second and then run it again and so on until the `action` returns "false". The `setup` and `action` functions are further explained in semantic rule 5.7 in section 5.1.

5.2.1.1 Graphical User Interface

A part of the built-in functionality is the graphical user interface (GUI).

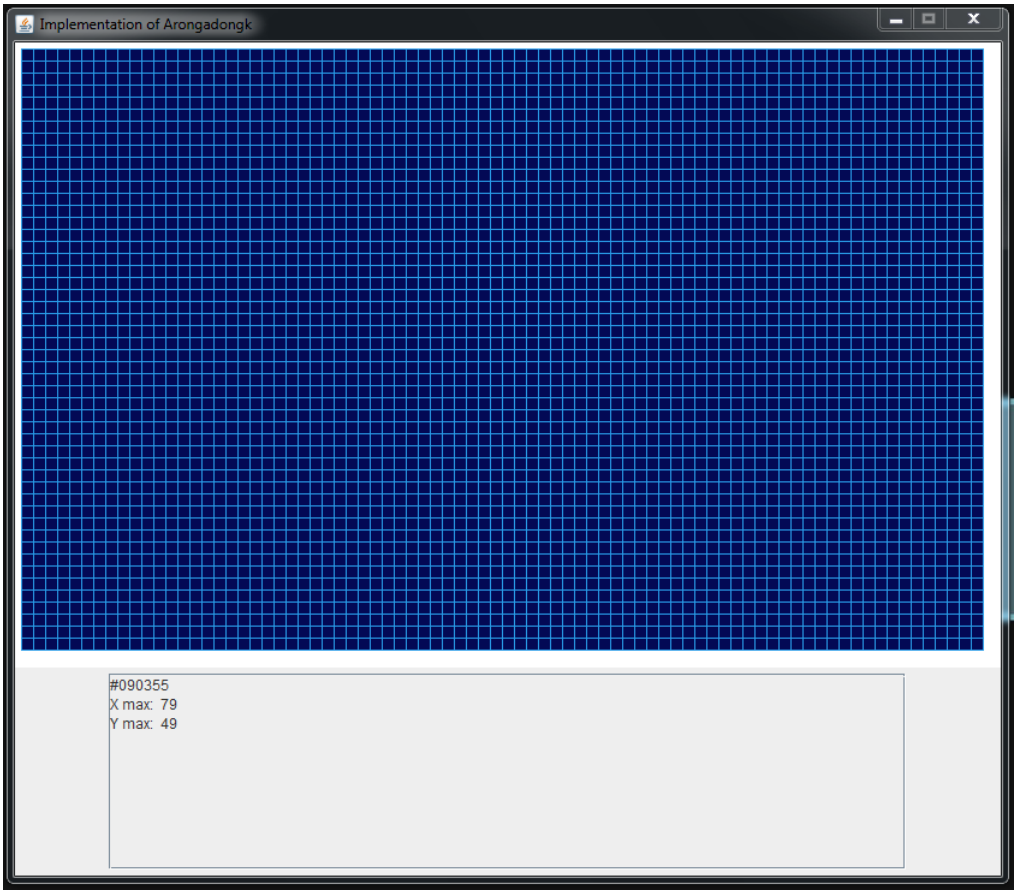


Figure 5.2: *The GUI of an application written in Arongadongk.*

The GUI consists of a grid and a console. An example is seen in figure 5.2. The grid consists of a number of cells, and a border that surrounds them. The individual cells' color can be changed by an API call, e.g. in order to display the location of an agent. It is possible to write to the console by using different API calls.

5.2.2 Application Programming Interface

A developer writing in Arongadongk does not need to worry about implementing a GUI, but it restricts him from making fundamental changes to it. He can however interact with the GUI to some extent with API calls.

While supplying functions for the GUI and the console the language needs to provide the user with some common functions, e.g. a random number generator. The programming language needs to supply the user with some common functions. All this can be referred to as Arongadongk's *API*. Arongadongk's API provides different commonly used functions, so the programmer does not need to implement them every time he needs to use them. Some functions provide a "link" to some of the elements, which are otherwise out of the programmer's control. A list of the key functions in Arongadongk can be found in figure 5.3. Some functions have been omitted from the figure, because they are mundane or very similar to functions already in the list, these include, but are not limited to `randomBool`, `randomFloat`, and `floatToString`.

Function	Description
`setGridColor(int x, int y, string color)`	Sets the grid color of the specified coordinate to color.
`getGridColor(int x, int y)`	Returns the color of the specified coordinate.
`println(string input)`	Takes a string input and prints it to the console and ends with a newline character.
`print(string input)`	Takes a string input and prints it to the console without a newline character.
`randomInt(int min, int max)`	Generates and returns a random number between min and max.
`setGridBorderColor(string color)`	Sets the border color of the grid to color.
`setDelay(int n)`	Sets delay between `action` to n in milliseconds.
`intToString(int a)`	Takes the int a and converts it into a string.
`getHeight()`	Returns the height of the grid as an int.
`getWidth()`	Returns the width of the grid as an int.
`clearGrid()`	Clears/resets the grid to its state before `setup`.
`addInt(int i)`	Adds the integer, i, to a static list and then shuffles the list.
`getInt()`	Returns and removes the element at index 0 in the static list.

Figure 5.3: *A list of the key functions in Arongadongk's API.*

The functions `addInt` and `getInt` are added to Arongadongk's API due to the desire of randomizing the order of the function calls in action. The function `addInt` takes an integer and adds it to a list implemented in Java. This list is then shuffled with Java's built-in shuffle function. To get the numbers again the function `getInt` is used. When `getInt` is called it removes the element at index 0 and returns it. The idea is that each function is assigned a number, then that number is added to the list. A sort of loop is then implemented, and `getInt` is looped until it returns -1, as this means the list is empty. Every time it is looped through the number returned is checked and the function with

the corresponding number is run. Since Arongadongk does not have any sort of array this is an easy way to get the functionality of shuffling the different functions the developer want run. The two functions can potentially be used for other things, but is originally thought to only shuffle functions. These two functions can be used to randomize the order in which the different agents' actions in a multi agent system are executed.

5.2.3 Creating an Application in Arongadongk

Usage of the functionality and the API calls described in section 5.2.1 and 5.2.2 is essential when creating an application in Arongadongk. The `setup` function is run once, and its purpose is to set up the structure of the application. Setting the grid to the developer's preference can be done by API calls. Initializing agents can be done by creating a function, or simply having all the required code in the `setup` function. The idea is that agents could be represented by coloring one or more cells in the grid. Instead of searching for a specific color in the grid, global variables can be used to store an agent's position.

The `setup` and `action` functions are boolean functions. If they return "false", the program will terminate or end respectively. If `setup` returns "true", `action` will be run as long as it returns "true". Making `setup` return "false" can be used to indicate that an error has occurred, which would mean that it would not make sense to run the application and thus the program terminates.

The `action` function is used to perform a single step of the program. Whatever happens is up to the developer, but agents could move, obstacles could appear, agents could die etc.

If `action` returns "true", it is called again. If `action` returns "false", the program stops executing without terminating. This allows the developer to see the final result.

It is therefore important to make sure `action` returns "false" at some point if the application is not supposed to run indefinitely.

Each agent can be represented as a function that is called in the `action` function.

There is currently no way to implement multiple similar agents without repeating code. This is due to the current limitations of Arongadongk, since a similar agent must be using either the same function or a similar one. As the movement of an agent is based on various global variables each agent will need its own function.

Having all this functionality, which is generally useful for creating a multi agent system, saves the developer time. Instead of having to implement a GUI and features for using it, the developer can focus on creating the multi agent system.

5.3 Code Generation

The purpose of code generation is to apply the semantic rules, specified in the Semantic Rules section (5.1), and translate the representation of the Arongadongk source program – expressed as an AST – into the target language chosen in the Language Choices section (2.1), namely Java, while also expressing the functionality explained in the Functionality & API section (5.2).

5.3.1 Java Encoding

As with type checking we have chosen to implement a visitor to traverse the decorated abstract syntax tree of the Arongadongk source program, while encoding into Java. Our **JavaEncoder** class constructor needs a so called "emitter", which takes care of the output of the Java encoder with two methods, namely `emit` and `emitln`. These are implemented by the emitter with other auxiliary functionality, such as taking care of printing scope levels correctly and with indentation.

Separating the encoding functionality from the emitting functionality in this way allows us to easily alter the way we use the result of the compilation, such as either outputting to the screen or writing to a file, etc.

An example of applying a semantic rule – [FUNCTION$_{\text{S-DECL}}$] rule 5.8 – is shown in code snippet 5.1. [FUNCTION$_{\text{S-DECL}}$] specifies that the function environment gets altered with the new function. This is expressed in code snippet 5.1 where a Java method with the `private` and `static` keywords are created. Since Arongadongk is an imperative language, we have chosen to keep all methods, globals, and constants static.

```
1  public Object visitFunctionDeclaration(FunctionDeclaration decl,
       Object arg) {
2      emitter.printIndent();
3      emitter.emit("private static " + JavaCoercer.getJavaType(decl.
           returnType) + " " + JavaCoercer.getVarName(decl) + "(");
4      decl.parameterList.visit(this, null);
5      emitter.emitln(")");
6
7      if(decl.singleStatement instanceof ImbracedStatement)
8      {
9          // ImbracedStatement opens and closes scopes by itself
10         decl.singleStatement.visit(this, null);
11     } else {
12         //Since we want to translate to java, we need to
13         //make braces
14         emitter.printIndent();
15         emitter.emitln("{");
16         emitter.openScope();
17         decl.singleStatement.visit(this, null);
18         emitter.closeScope();
19         emitter.printIndent();
20         emitter.emitln("}");
21     }
22     return null;
23 }
```

Code snippet 5.1: *The `visitFunctionDeclaration` method.*

5.3.2 Java Coercer

Code snippet 5.1 also illustrates the usage of the **JavaCoercer** class, which sole purpose is to translate the Arongadongk types into compatible Java types, and to translate the Arongadongk variable names declared in the Arongadongk source program into names that are compatible in Java.

Since Arongadongk supports having variable names like "instanceof" – which is a keyword in Java – the Arongadongk source program variable names cannot be used directly in Java. As Arongadongk allows to hide previously declared variables in deeper scopes, we have to take precautions when encoding within Arongadongk scopes, since some cases of Arongadongk scopes are not recognized as scopes in Java, e.g. if-statements open a new scope in Arongadongk where this is not the case in Java.

Our solution is to use the prefix "agd_" for all variables, and use the postfix "_x" – where x denotes the Arongadongk scope level for all variables that are declared at other scope levels than zero. The prefix ensures that no variable declared in Arongadongk can be confused with a java keyword such as `volatile` or `synchronized` [2], etc. An example of this is shown in code snippet 5.2 and what it compiles to in code snippet 5.3. For an elaboration on the scope rules see section 4.2.

```
1  int foo()
2  {
3      int bar;
4      if (1 == 1)
5      {
6          // Following is valid in Arongadongk
7          int bar;
8      }
9      return 0;
10 }
```

Code snippet 5.2: *Example of allowed variable declarations in Arongadongk scopes.*

```
1  private static int agd_foo_1()
2  {
3      int agd_bar_3 = 0;
4      if (1 == 1)
5      {
6          int agd_bar_5 = 0;
7      }
8      return 0;
9  }
```

Code snippet 5.3: *Result after compiling code snippet 5.2.*

The examples in code snippet 5.2 and code snippet 5.3 illustrate how we make sure that our Arongadongk variables do not get Java errors such as "The local variable agd_bar_3 may not have been initialized". The Arongadongk compiler does not throw an error when a declared variable is read, instead we choose to implicitly instantiate the variable. How this is taken care of can be seen in code snippet 5.4.

```
1  public Object visitVariableDeclarationStatement(
2          VariableDeclarationStatement state, Object arg) {
3      emitter.printIndent();
4      emitter.emit(JavaCoercer.getJavaType(state.type) + " "+
           JavaCoercer.getVarName(state));
5
6      if(state.getType() == StandardEnvironment.
           getBoolTypeDeclaration()) {
7          emitter.emit(" = false");
8      } else if(state.getType() == StandardEnvironment.
           getIntTypeDeclaration()) {
9          emitter.emit(" = 0");
10     } else if(state.getType() == StandardEnvironment.
           getFloatTypeDeclaration()) {
11         emitter.emit(" = 0.0");
12     } else if(state.getType() == StandardEnvironment.
           getStringTypeDeclaration()) {
13         emitter.emit(" = \"\"");
14     }
15
16     emitter.emitln(";");
17     return null;
18 }
```

Code snippet 5.4: *The* ***visitVariableDeclarationStatement*** *method in the* ***JavaEncoder*** *class.*

Chapter 6

Epilogue

This chapter evaluates Arongadongk and presents the improvements that are left out due to lack of time as well as a conclusion on the problem statement. The improvement section contains different kinds of improvements, from adding types to changing programming paradigms.

6.1 Evaluation of Arongadongk

Since we have implemented a compiler for our language within the given time-frame we consider our language implementable.

Since we allow variables to be hidden we believe that our language is writable, due to the fact that the programmer using Arongadongk does not need to come up with a lot of different names for e.g. temporary variables. Furthermore, we do not have long keywords (the longest being "global"), which should enhance writability as well.

Readability might have suffered because variables can be hidden. A good overview is required to know which declaration a variable belongs to if the same variable name is used several places.

The reliability is considered acceptable, since we have a strongly typed type system. However, since we initialize each variable it affects the reliability negatively because variables can be used before the programmer initializes them. This can create unintended situations.

Orthogonality has not been taken care of very well. We take orthogonality into account in expressions since we allow combinations of different categories of operators, such as logical and comparison operators, to be used in one calculation. This increases writability, and possibly readability, because it can be difficult to combine several smaller expressions into the exact meaning of the entire represented expression.

The cost for learning Arongadongk is low as long as the programmer learning is familiar with the C-family.

6.2 Improvements

When developing in a new language, we learn what it lacked, and what should be removed. Unfortunately, time is limited so the improvements we would like

to implement will be described in the following sections.

6.2.1 General Language Improvement

When designing a programming language there are sometimes solutions that may seem better than others, due to how difficult and time consuming they are to implement. In the current version of the Arongadongk language it is not possible to declare and initialize a variable in one statement. This makes the BNF simpler, but it increases the time a developer has to use to write an application. Small improvements like these are not seen as necessary before development in the language starts. Another improvement could be to increment a variable by 1 or an arbitrary number. With Arongadongk we decided to only be able to assign a variable an expression with no shortcuts. This means that it is not possible to increment a variable with the usual C-family syntax "`++`", as well as adding 5 to a variable with the shortcut "`+= 5`".

These two improvements are not particularly difficult to implement, but changing the syntax and semantics to support them can be time consuming. The declaration and initialization at the same time requires a change to the BNF and thereby also the parser.

Further improvements could be adding additional kinds of loops like a for loop or a do-while loop. It can be argued that a for loop can easily be imitated by a while loop, but due to the fact that the C-family generally implement a for loop in their syntax, we believe that it would be a good addition for a higher level of writability.

6.2.2 Arrays

For many languages arrays or lists are standard data types – but not for Arongadongk. Programming without arrays is for many tasks tedious and time consuming. Imagine that an Arongadongk program is written to simulate a colony of tortoises and their behavior, every tortoise needs a position, given by an x and a y-coordinate. This would require two variables for each tortoise, instead of two lists; one for y-coordinates and one for x-coordinates.

6.2.2.1 Syntax

To implement arrays the syntax must be changed in order to allow declaration of array types. We need to change several rules in the grammar in EBNF seen in section 3.2.

The global declaration is changed to:

$$\textit{single-declaration} ::= \textbf{global}\ \textit{identifier}(\ [\ \textit{intLiteral}\ ]\ )^*\ \textit{identifier}\ \textbf{;} \quad (6.1)$$

The declaration-statement is changed to:

$$\textit{single-statement} ::= \textit{identifier}(\ [\ \textit{intLiteral}\ ]\ )^*\ \textit{identifier}\ \textbf{;} \quad (6.2)$$

The assignment-statement is changed to:

$$\textit{single-statement} ::= \textit{identifier}(\ [\ \textit{expression}\ ]\ )^*\ =\ \textit{expression}\ \textbf{;} \quad (6.3)$$

The variable-factor rule is changed to:

$$factor ::= identifier(\ [\ expression\ ]\ ) * \tag{6.4}$$

Two new tokens are added, namely the left and right square brackets. We choose to have an expression within the square brackets ("[" and "]"), this allows for code as in code snippet 6.1.

```
1 global int[] Arr;
2 int foo(){
3     int[][] A;
4     A[0][A[1][2]] = 4;
5     Arr[2] = 2;
6     return Arr[0];
7 }
```

Code snippet 6.1: *Syntacticly valid Arongadongk code with arrays.*

6.2.2.2 Contextual Constraints

The code snippet 6.1 is syntacticly valid, but we need some constraints for the arrays to work. The expression within the square brackets must evaluate to an integer and composite types must be added. The composite type is denoted by $A[\![\tau]\!]$. To make type rules and semantics for arrays the abstract syntax used in section 4.1.1 must be altered. The following rules are added:

$$x ::= x_1[e] \tag{6.5}$$

$$t ::= t_1[il] \tag{6.6}$$

It appears that any variable or type must be an array, but it means that a variable can be a variable or an array.
For x we add a new rule:

[ARRAY$_{\text{EXP}}$]

$$\frac{E \vdash x : \tau \quad e = Int}{E \vdash \ x[e]\ : A[\![\tau]\!]} \tag{6.7}$$

For t we add a new rule:

[ARRAY$_{\text{TYPE}}$]

$$\frac{E \vdash t : \tau}{E \vdash \ t[il]\ : A[\![\tau]\!]} \tag{6.8}$$

6.2.2.3 Semantics

For the arrays to work we need to define what e.g. $int[3]$ actually implies. The only thing needed to be changed is the variable environment. When the environment saves an array it should create i new locations, where i is the value of the integer literal within the square brackets in a declaration. When accessing the variables within an array by using e.g. $A[2]$, the environment should get the location of A plus an offset of two $(env_v(A) + 2)$.

6.2.2.4 Implementation

Arrays can be implemented with Java's arrays or its **ArrayList** class. If ArrayLists are used the integer literal in the declaration is unnecessary.

6.2.3 Graphical User Interface Improvements

In the current version of the Arongadongk compiler a developer coding in Arongadongk is incapable of making any major changes to the GUI. This means that the grid is consistently 80 by 50 cells and cells are 10 pixels by 10 pixels in size. The console beneath the grid is consistently 10 rows and 60 columns in size.

It is not unlikely that an application were to require a larger grid i.e. a grid with more cells. Following that logic, it would not be unlikely that the cells were required to be smaller in size in order to keep the overall size of the application to a size that is easy to handle. It may not be preferable to have an application that is not resizable, if the application fills the entire display.

6.2.4 Improved Multi Agent System Support

As it is explained in section 1.2, a key feature is to have the agents interact with each other, this could be through mechanisms like voting (section 1.2.2.1), auctions (section 1.2.2.2), or cooperative learning (section 1.2.2.3). Implementing these features would be an improvement to Arongadongk. If we were to add communication between agents, a small-step-semantic is preferred over an big-step-semantic, since this is supported in small-step-semantic [4, chap. 8]. Due to insufficient time, we have not considered what it would take to implement these.

6.2.5 Object-oriented Programming Paradigm

Since the current paradigm of Arongadongk does not support object-oriented programming, which we think will be very suitable for agents, it is an obvious choice to implement the object-oriented programming paradigm in Arongadongk. The object-oriented programming paradigm will improve the implementation of several similar agents.

6.3 Conclusion

Our motivation led us to research multi agent systems and define our criteria, which led us to the following problem statement:

> *Given our time frame, how can we design a programming language and implement a compiler that relieves the programmer from trivial development tasks when creating a multi agent system?*

In order to start the process of answering our problem statement, we decided that we wanted an imperative sequential language, along with a low number of

types and operators. We ended up choosing to make a C-family like LL(1) CFG, which allowed us to make a recursive-descent parser responsible for constructing an AST. Static scope rules were made to allow the programmer to hide all declarations except functions. This heightened the level of writability, but may have lowered readability. We made contextual constraints, which ensure that all functions will return eventually and that they will return a value of the correct type. This allowed us to implement a checker, which does not allow compilation of incorrectly typed programs. Furthermore, our language and compiler support calls to functions that are declared later in the source code than the call.

We defined the functionality, which has the purpose of relieving the programmer of trivial development tasks, by creating a built-in GUI and code that handles the automation of the iterative calls to the `action` function in the source code.

The functionality combined with the semantic rules were implemented in our code generator that generates Java code. This Java code is then compiled into Java bytecode, which can run on the Java Virtual Machine, which is commonly known to be platform independent. Had we chosen a larger scope for our project we should have improved our language and compiler with additional built-in functionality and extensions such as arrays, additional multi agent system mechanisms, and object orientation.

Bibliography

[1] TIOBE Software BV. Tiobe programming community index for april 2011. Webpage, April 2011. URL `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Last viewed: 2011-04-14.

[2] Neil Coffey. The volatile keyword in java. Webpage. URL `http://www.javamex.com/tutorials/synchronization_volatile.shtml`. Last viewed: 2011-05-25.

[3] Éric Lévénez. Computer languages history. Webpage. URL `http://www.levenez.com/lang/`. Last viewed: 2011-05-25.

[4] Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010. ISDN: 978-0-521-14709-5.

[5] Oracle. Java inheritance. Webpage, . URL `http://download.oracle.com/javase/tutorial/java/IandI/subclasses.html`. Last viewed: 2011-05-03.

[6] Oracle. Java^{TM} platform, standard edition 6 overview. Webpage, . URL `http://download.oracle.com/javase/6/docs/technotes/guides/index.html`. Last viewed: 2011-05-20.

[7] Robert W. Sebesta. *Concepts of Programming Languages*. PEARSON, international ninth edition, 2010. ISBN-13: 978-0-13-246558-8.

[8] Inc Sun Microsystems. The java language specification, third edition. WWW, 2005. URL `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`. Last viewed: 2011-05-13.

[9] Herb Sutter. The c family of languages: Interview with dennis ritchie, bjarne stroustrup, and james gosling. Webpage, 2009. URL `http://www.gotw.ca/publications/c_family_interview.htm`. Last viewed: 2011-05-25.

[10] TIGCC Team. C language keywords. Webpage. URL `http://tigcc.ticalc.org/doc/keywords.html#constl`. Last viewed: 2011-05-24.

[11] José M Vidal. *Fundamentals of Multiagent Systems*. 2010. URL `http://jmvidal.cse.sc.edu/papers/mas.pdf`. Last viewed: 2011-04-14.

[12] David A Watt and Deryck F Brown. *Programming Language Processors in Java*. Pearson Educational Limited, 2000. ISBN-13:978-0-13-025786-4.

[13] David A Watt and Deryck F Brown. Triangle tools. Webpage, 2003. URL http://www.dcs.gla.ac.uk/~daw/books/PLPJ/. Last viewed: 2011-05-03.

Appendix

Appendix A

Type Rules

[$\textsc{Type}_{\text{T}}$]

$$E \vdash t : \tau \tag{A.1}$$

$$\text{where} \qquad E(t) = \tau$$

A.1 Statements

[$\textsc{Subsumption}_{\text{M-STM}}$]

$$\frac{E \vdash MS : \tau_1 \quad \tau_1 <: \tau_2}{E \vdash MS : \tau_2} \tag{A.2}$$

A.2 Declarations

[$\textsc{Global}_{\text{DECL}}$]

$$\frac{E \vdash t : \tau \quad E \vdash D : \textsc{Ok}}{E \vdash \textbf{global}\ t\ x\ \textbf{;} D : \textsc{Ok}}$$

[$\textsc{Empty}_{\text{DECL}}$]

$$E \vdash \varepsilon : \textsc{Ok}$$

A.3 Parameter Lists

A proper formal parameter list is used whenever multiple parameters are needed, thus it has a leading comma. A proper formal parameter list is well-typed if the type name evaluates to a type and its proper formal parameter list is well-typed as well. The types are, as with the formal parameter list, concatenated.

[ACTUAL$_{\text{PFPL}}$]

$$\frac{E \vdash t : \tau_1 \quad E \vdash pfpl : \tau_2}{E \vdash \mathbf{,}\ t\ x\ pfpl : \tau_1 \times \tau_2} \tag{A.3}$$

A proper actual parameter list is used for having multiple parameters when calling a function, thus the leading comma. As with the actual parameter list the proper actual parameter list is well-typed if its expression evaluates to a type, and its proper actual parameter list is well-typed.

[ACTUAL$_{\text{PAPL}}$]

$$\frac{E \vdash e : \tau_1 \quad E \vdash papl : \tau_2}{E \vdash ,\ e\ papl : \tau_1 \times \tau_2} \tag{A.4}$$

A.4 Expressions

[UNARY-NEGATION$_{\text{EXP}}$]

$$\frac{E \vdash e : \tau \quad \tau = Float}{E \vdash -\ e\ : \tau} \tag{A.5}$$

[UNARY-NOT$_{\text{EXP}}$]

$$\frac{E \vdash e : \tau \quad \tau = Bool}{E \vdash \mathbf{!}\ e\ : \tau} \tag{A.6}$$

Appendix B

Ensure Return Value of Functions

[While$_{\text{STM}}$]

$$\frac{S : r}{\textbf{while}(e)\ S : \mathit{ff}} \quad \text{if} \quad r \neq \mathit{error} \tag{B.1}$$

[While-error$_{\text{STM}}$]

$$\frac{S : \mathit{error}}{\textbf{while}(e)\ S : \mathit{error}} \tag{B.2}$$

[Function-call$_{\text{STM}}$]

$$f\,(\ \mathit{apl}\ )\ \textbf{;} : \mathit{ff} \tag{B.3}$$

[Assignment$_{\text{STM}}$]

$$x = e\ \textbf{;} : \mathit{ff} \tag{B.4}$$

[Local-Declaration$_{\text{STM}}$]

$$t\ x\ \textbf{;} : \mathit{ff} \tag{B.5}$$

Appendix C

Semantics

C.1 Declarations

[GLOBAL$_{\text{DECL}}$]

$$\frac{\langle D, env_f, env_v'', sto\rangle \xrightarrow[decl]{} (env_f', env_v', sto')}{\langle t\ x\ ;D, env_f, env_v, sto\rangle \xrightarrow[decl]{} (env_f', env_v', sto')} \tag{C.1}$$

where $l = env_v(next)$

and $env_v'' = env_v\,[x \mapsto next]\,[next \mapsto new(l)]$

C.2 Parameters

[EMPTY$_{\text{FPL}}$]

$$\langle \varepsilon, env_v\rangle \xrightarrow[fpl]{} env_v \tag{C.2}$$

[EMPTY$_{\text{PFPL}}$]

$$\langle \varepsilon, env_v\rangle \xrightarrow[pfpl]{} env_v \tag{C.3}$$

[EMPTY$_{\text{APL}}$]

$$\langle \varepsilon, env_v\rangle \xrightarrow[apl]{} env_v \tag{C.4}$$

[EMPTY$_{\text{PAPL}}$]

$$\langle \varepsilon, env_v\rangle \xrightarrow[papl]{} env_v \tag{C.5}$$

[ACTUAL$_{\text{PFPL}}$]

$$\frac{\langle pfpl, env_v[x \mapsto next][next \mapsto newl(l)]\rangle \xrightarrow[pfpl]{} env'_v}{\langle\ ,\ \ t\ f\ pfpl, env_v, \rangle \xrightarrow[pfpl]{} env'_v} \tag{C.6}$$

$$\text{where } l = env_v(next)$$

[ACTUAL$_{\text{PAPL}}$]

$$\frac{env_f, env_{vf}, env_{va} \vdash \langle papl, pfpl, sto''[l \mapsto v]\rangle \xrightarrow[papl]{} sto'}{env_f, env_{vf}, env_{va} \vdash \langle\ ,\ e\ papl,\ ,\ t\ x\ pfpl, sto\rangle \xrightarrow[papl]{} sto'} \tag{C.7}$$

$$\begin{aligned} \text{where} \quad & env_f, env_{va}, sto \vdash e \xrightarrow[exp]{} (v, sto'') \\ \text{and} \quad & l = env_{vf}(x) \end{aligned}$$

C.3 Statements

[IF-ELSE-FALSE$_{\text{STM}}$]

$$\frac{env_f \vdash \langle S_2, env_v, sto'', ret\rangle \xrightarrow[stm]{} (env'_v, sto', ret')}{env_f \vdash \langle \textbf{if (}\ e\ \textbf{)}\ S_1\ \textbf{else}\ S_2, env_v, sto, ret\rangle \xrightarrow[stm]{} (env_v, sto', ret')} \tag{C.8}$$

$$\text{if} \qquad env_f, env_v \vdash \langle e, sto\rangle \xrightarrow[exp]{} (ff, sto'')$$

[IF-TRUE$_{\text{STM}}$]

$$\frac{env_f \vdash \langle S, env_v, sto'', ret\rangle \xrightarrow[stm]{} (env'_v, sto', ret')}{env_f \vdash \langle \textbf{if (}\ e\ \textbf{)}\ S, env_v, sto, ret\rangle \xrightarrow[stm]{} (env_v, sto', ret')} \tag{C.9}$$

$$\text{if} \qquad env_f, env_v \vdash \langle e, sto\rangle \xrightarrow[exp]{} (tt, sto'')$$

[IF-FALSE$_{\text{STM}}$]

$$env_f \vdash \langle \textbf{if (}\ e\ \textbf{)}\ S, env_v, sto, ret\rangle \xrightarrow[stm]{} (env_v, sto', ret) \tag{C.10}$$

$$\text{if} \qquad env_f, env_v \vdash \langle e, sto\rangle \xrightarrow[exp]{} (ff, sto')$$

[WHILE-TRUE-CONTINUE$_{\text{STM}}$]

$$\frac{\begin{array}{c} env_f \vdash \langle S, env_v, sto''', ret\rangle \xrightarrow[stm]{} (env''_v, sto'', ret'') \\ env_f \vdash \langle \textbf{while (}\ e\ \textbf{)}\ S, env''_v, sto'', ret''\rangle \xrightarrow[stm]{} (env'_v, sto', ret') \end{array}}{env_f \vdash \langle \textbf{while (}\ e\ \textbf{)}\ S, env_v, sto, ret\rangle \xrightarrow[stm]{} (env_v, sto', ret')} \tag{C.11}$$

$$\begin{aligned} \text{if} \quad & env_f, env_v \vdash \langle e, sto\rangle \xrightarrow[exp]{} (tt, sto''') \\ \text{and} \quad & ret(return) = ff \end{aligned}$$

[WHILE-TRUE-RETURN$_{\text{STM}}$]

$$env_f \vdash \langle \textbf{while (}\ e\ \textbf{)}\ S, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v, sto, ret) \tag{C.12}$$

$$\text{if} \qquad ret(return) = tt$$

[WHILE-FALSE$_{\text{STM}}$]

$$env_f \vdash \langle \textbf{while (}\ e\ \textbf{)} S, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v, sto', ret) \tag{C.13}$$

$$\text{if} \qquad env_f, env_v \vdash \langle e, sto\rangle \underset{exp}{\longrightarrow} (\mathit{ff}, sto')$$

[ASSIGNMENT$_{\text{STM}}$]

$$env_v \vdash \langle x = e\textbf{;}, env_v, sto, ret\rangle \underset{stm}{\longrightarrow} (env_v, sto'[l \mapsto v], ret) \tag{C.14}$$

$$\begin{aligned} \text{where} \quad & env_f, env_v \vdash \langle e, sto\rangle \underset{exp}{\longrightarrow} (v, sto') \\ \text{and} \quad & env_v(x) = l \end{aligned}$$

C.4 Expression

C.4.1 Multiple Expressions

[MINUS$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 - e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.15}$$

$$\text{where} \quad v = v_1 - v_2$$

[MULTIPLICATION$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 * e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.16}$$

$$\text{where} \quad v = v_1 \cdot v_2$$

[DIVISION$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 / e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.17}$$

$$\begin{aligned} \text{if} \quad & v_2 \neq 0 \\ \text{where} \quad & v = \tfrac{v_1}{v_2} \end{aligned}$$

[LESS$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 < e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.18}$$

$$\text{where} \quad v = v_1 < v_2$$

[GREAT$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 > e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.19}$$

$$\text{where} \quad v = v_1 > v_2$$

[NOT-EQUAL$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 \texttt{ != } e_2, sto\rangle \underset{exp}{\longrightarrow} (v_1 \neq v_2, sto'')} \tag{C.20}$$

$$\text{where} \qquad v = \begin{cases} \mathit{ff} & \text{if} \quad v_1 = v_2 \\ \mathit{tt} & \text{if} \quad v_1 \neq v_2 \end{cases}$$

[EQUAL$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 == e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.21}$$

$$\text{where} \qquad v = \begin{cases} \mathit{tt} & \text{if} \quad v_1 = v_2 \\ \mathit{ff} & \text{if} \quad v_1 \neq v_2 \end{cases}$$

[LESS-OR-EQUAL$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 <= e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.22}$$

$$\text{where} \quad v = v_1 \leq v_2$$

[GREAT-OR-EQUAL$_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 >= e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.23}$$

$$\text{where} \quad v = v_1 \geq v_2$$

[OR_{EXP}]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 \ \texttt{||}\ e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.24}$$

$$\text{where} \quad v = v_1 \vee v_2$$

[AND_{EXP}]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto') \quad env_f, env_v \vdash \langle e_2, sto'\rangle \underset{exp}{\longrightarrow} (v_2, sto'')}{env_f, env_v \vdash \langle e_1 \ \&\& \ e_2, sto\rangle \underset{exp}{\longrightarrow} (v, sto'')} \tag{C.25}$$

$$\text{where} \quad v = v_1 \wedge v_2$$

C.4.2 Single Expressions

[$\text{ARITHMETIC-NEGATION}_{\text{EXP}}$]

$$\frac{env_f, env_v \vdash \langle e_1, sto\rangle \underset{exp}{\longrightarrow} (v_1, sto')}{env_f, env_v \vdash \langle -e_1, sto\rangle \underset{exp}{\longrightarrow} (v, sto')} \tag{C.26}$$

$$\text{where} \quad v = -v_1$$

[$\text{BOOLEAN-FALSE}_{\text{EXP}}$]

$$env_f, env_v \vdash \langle bl, sto\rangle \underset{exp}{\longrightarrow} (\mathit{ff}, sto) \qquad \text{if} \qquad bl = \textbf{false} \tag{C.27}$$

[$\text{FLOAT}_{\text{EXP}}$]

$$env_f, env_v \vdash \langle fl, sto\rangle \underset{exp}{\longrightarrow} (v, sto) \qquad \text{if} \qquad v = \mathcal{F}\,(\mathit{fl}) \tag{C.28}$$

[$\text{STRING}_{\text{EXP}}$]

$$env_f, env_v \vdash \langle sl, sto\rangle \underset{exp}{\longrightarrow} (v, sto) \qquad \text{if} \qquad v = \mathcal{S}\,(sl) \tag{C.29}$$