LagNa
- a programming language for GamNa

DAT2 project, Spring 2004.

By projectgroup E4-213

Aalborg University
Department of Computer Science

Aalborg University
Department of Computer Science

TITLE:
LagNa
- a programming language
for GamNa

THEME:
Tools, languages and translators

PROJECT PERIOD:
2/2-28/5, 2004

PROJECT GROUP:
E4-213

GROUP MEMBERS:
Esben Bach
Lars H. Jensen
Marc N. Kammersgaard
Andreas B. H. Madsen
Jasper Melsvik
Rasmus B. Nielsen
Bjørn M. Vester

SUPERVISOR:
John Knudsen

SYNOPSIS:

In this project we develop a programming language for the game GamNa. This is done using Context Free Grammars and operational semantics. Furthermore we develop a verification system following an idea by C. A. R. Hoare. And implement a compiler for the language using SableCC. An important aspect in the language design is easy read- and write-ability, we conclude that this has been achieved.

NUMBER OF COPIES: 9

NUMBER OF PAGES: 142

CONCLUDED: 28/5-2004

The sourcecode of the engine GamNa and of the compiler for LagNa can be found at:

http://www.cs.auc.dk/ esben/dat2/lagna/

It will be available on that page at least until the year 2007.

Esben Bach

Lars Hornbæk Jensen

Marc Nyboe Kammersgaard

Andreas Bue Holmgaard Madsen

Jasper Melsvik

Rasmus Bang Nielsen

Bjørn Mølgård Vester

Contents

Chapter 1

Introduction

A programming language is a something which is made for the purpose of giving a higher abstraction level than writing computer instructions (binary code) or a lower level language. The purpose of this higher abstration level is to ease the writeability for the programmer, it allows the programmer to use instruction that can be translated to perhaps several lower level instructions. Code written in a high level language is also more understandable than programs written in lower level languages, since the higher up we get in programming language level they look more and more like normal languages. When creating a programming languages it is necessary to indicate how the languages should be written, how it must work, and finally a program which can translate the language into instructions which the computer can understand.

The objective of this project is to develop such a higher level programming language, which we will call LagNa. A special purpose langauge used for the development of bots for the game GamNa [Tea]. The language should be developed using both a well-formed syntax and semantics. Furthermore we will look into the possibility of doing program verifikation and we will implement a compiler for the language.

This project report will be structured into seven chapters: *Chapter 1* will give an overview of the game and the elements essential to the language, which are to be developed. These elements being the environment the bots are surrounded by, the engine that runs the game and the communication between the two. *Chapter 2* outlines the priorities of the language and its quality goals. *Chapter 3* contains the syntactic description of the language, and *Chapter 4* describes the operational semantics of the language. *Chapter 5* will describe how the type checking of LagNa will be done. In *Chapter 6* we develop an inference system for program verification, and *Chapter 7* describes the compiler.

1.1 The Game GamNa

The game takes place in an arena where multiple computer controlled entities called bots are fighting for life or death, without any user intervention. The game is over when a bot has defeated all the opposing bots. This might not seem like much of a game when there is no user interaction, but the aspect of the gameplay lies elsewhere, namely in the programming of the bots. The challenge of the game is to program the ”intelligence” of a bot in order for it to be *the last bot standing*. In short, the objective of the game is: “Code the superior bot”. This game has been developed specifically for this project, and is inspired by [Alp].

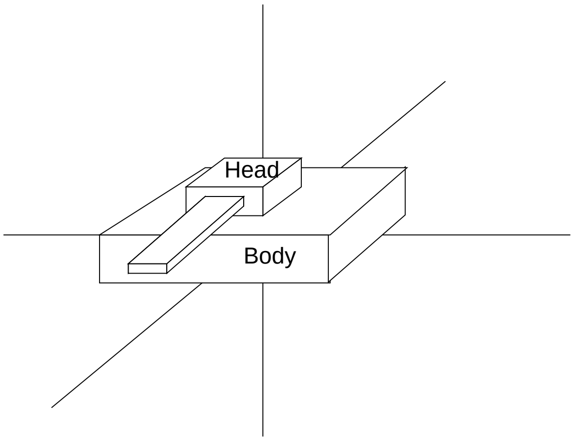


Figure 1.1: A bot and its main components visualized in simple 3D. In this figure we have depicted a bot as a tank.

1.1.1 Description of GamNa

Before we can create a language for the game, we need to know a little about the characteristics of the game. This includes the environment, in which the bots battle, and what capabilities they have in the game.

Arena This is the arena where the battle takes place. It is a two-dimensional rectangular environment, which size differs randomly from battle to battle. An arena contains randomly placed obstacles and is surrounded by a wall, which limits the battle to a small enclosure. An example of a battelfield from GamNa can be seen on Figure 1.1

Obstacles An obstacle is an inactive static object. These obstacles restricts the sight of bots, and neither bullets nor bots can move through them. An examble of an obstacle in GamNa could be the cross on Figure 1.1

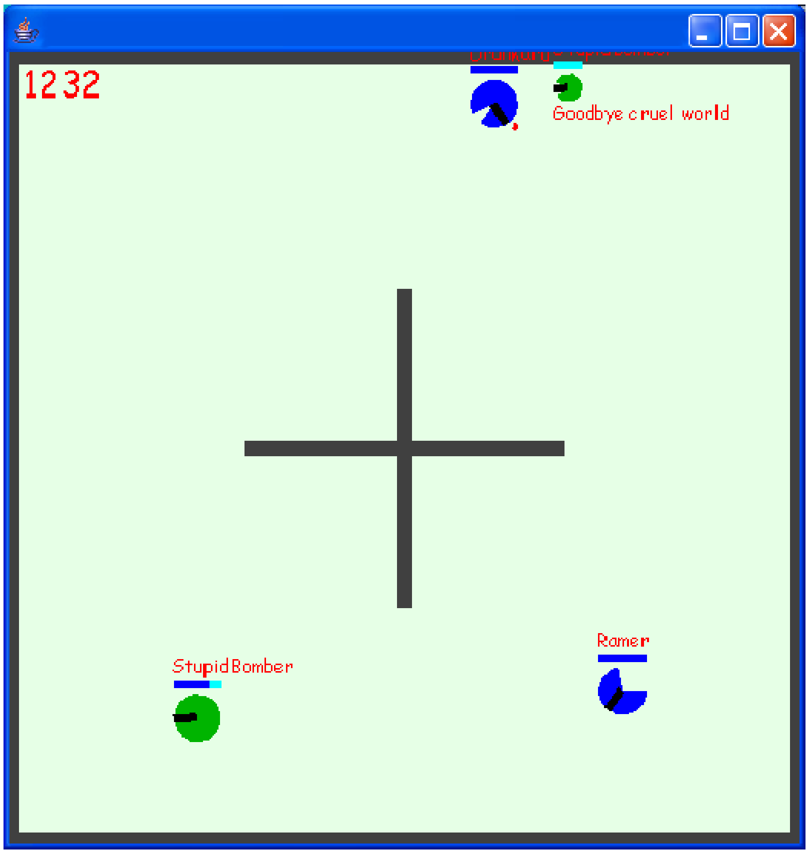
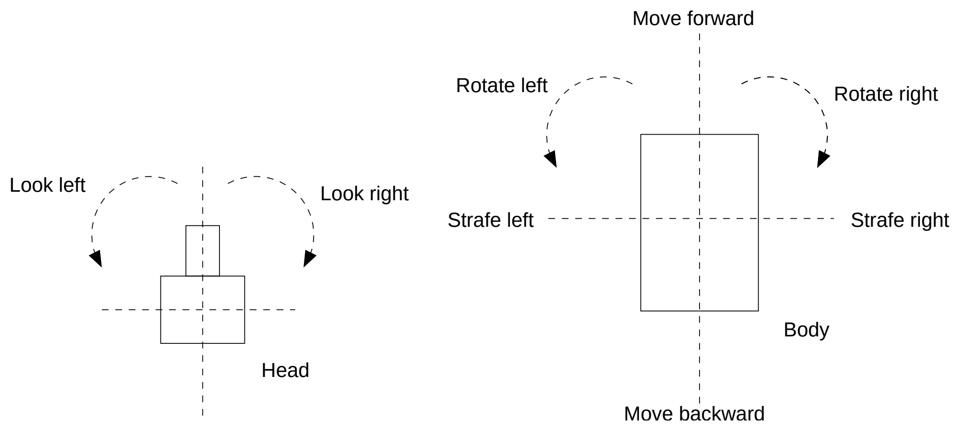


Figure 1.2: Screen shot of the arena from the game GamNa.

Bot This is a computer controlled, user programmed unit. A bot consists of two parts: a "body" and a "head". A bot is shown on Figure 1.1. The "body" controls the bot movement in the arena, and the "head" controls the bot vision. This makes the bot capable of moving in one direction while looking in another. The "body" and the "head" can rotate independently. A bot in GamNa looks like one of the "packmen" objects on Figure 1.1.

A bot can move in four directions: forwards, backwards, left and right. We will call this the bots movement axis, and it is equivalent to movement only along the x and y axis in a two dimensional plane. However, the movement axis can rotate, which we call "body rotation". Movement is at all times restricted to free areas of the arena. Movement and rotation of the body is shown on Figure 1.3(b) on the following page, and the rotation of the head is shown on Figure 1.3(a) on the next page.



(a) The rotation of the head. (b) Movement axis and rotation of the body.

Figure 1.3: Movement and rotation axis of the bots components.

Each bot has a straight line of sight that extends until an object is encountered. The sight is fixed to the rotation of the head.

Furthermore, bots carry weapons, such as a pistol and bombs, which can be used to inflict damage upon other bots. The firing direction is fixed to the head, just as line of sight. When a bot is hit it receives information, of what direction the attack came from.

The bots are also restricted by some "physical" laws, such as the time it takes to rotate, and at which speed they can move. Furthermore, each weapon has a firing delay (defined by the individual weapon), before the weapon can be fired again.

Finally, a bot has a fixed amount of health. Whenever a bot is damaged, health is reduced, and when a bot runs out of health, it is declared dead and is removed, by the engine, from the arena.

Teams Teams can be defined so that the individual bots can cooperate to defeat an enemy team. There is no way for the bots to communicate with teammates directly. Bots just get the additional information of wherever or not they are on the same team when they spot a bot.

1.2 Game Engine and Bot Communication

The game engine is responsible for informing the bots of changes in their ”world”. This means that the engine must somehow notify the bots of events that has occurred. The engine achieves this by invoking events on the bots, and the bots can then respond to these events. There are six events in the game and all bots should implement these. If they do not, the engine will ignore these unimplemented events, and therfor not tell the bot when one of the events occur. All the events are described in Table 1.1 on page 5.

<table>
<tr><th>Event</th><th>Arguments</th><th>Description</th></tr>
<tr><td>OnCollision</td><td>real type, real collisionAngle.</td><td>Invoked whenever the bot collides with another object. Shown below is the values of the types and their meaning:
<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Obstacle</td></tr><tr><td>1</td><td>Bot</td></tr><tr><td>2</td><td>Bomb</td></tr></table>The collisionAngle, is the angle in degrees from where the bot collided to the “head direction”.</td></tr>
<tr><td>OnDeath</td><td>string nameOfKiller.</td><td>Invoked when the bot is dead.</td></tr>
<tr><td>OnHit</td><td>real hitAngle.</td><td>Invoked whenever a bot is damaged. The argument denote where the hit came from.</td></tr>
<tr><td>OnSight</td><td>real type, boolean isEnemy, real distance.</td><td>Invoked whenever a new object enters the bot’s “line of sight”.</td></tr>
<tr><td>OnStart</td><td></td><td>Invoked when the battle begins.</td></tr>
<tr><td>OnTargetHit</td><td></td><td>Invoked when a bot makes a successful attack.</td></tr>
</table>

Table 1.1: Event table.

The arguments shown on Table 1.1 holds information that a bot programmer can access. The ”type” argument tells us wherever we are dealing with an obstacle, a bot, or a bomb. The remaining arguments are trivially explained by their names.

1.2.1 Actions

Events covers the engine-to-bot communication. Since the engine is in charge of everything in the game, the bots needs a way to tell the engine what they want to do (bot-to-engine). This is accomplished by *actions*. Actions are special functions that control the movement, rotation, shooting, etc. of the bots. The bot programmers, can use these to control the behavior of their bots. The actions that are implemented in the game engine are shown in Appendix A on page 125.

1.3 Summary

The purpose of the language is to make it easy to develop bots for the game GamNa. From the description of the game, we get an idea to some guidelines for what the bots should be able to do within our language as a minimum. The API (Application Program(ming) Interface) described, outlines what the language should support.

1. All the events listed in Table 1.1.
2. All the actions listed in Appendix A on page 125

Chapter 2

Design

The previous chapter introduced the game and its parts, alongside a description of the engine, the communication with bots, and lastly some general requirements for the language. This chapter will deal with the design of the language and the parts needed to give us a language which fulfills our requirements.

2.1 Quality goals

The following section will focus on the quality goals for the language LagNa. The criteria used in this section have been written with inspiration from [Seb04, pages 8-20].

As described earlier the goal is to design a language which makes it easy to write bots for GamNa. This can be achieved through easy write- and readability. To ensure this, we have defined a set of quality goals to follow in the design process of LagNa.

The criteria we work from are listed in Table 2.1 on the following page, and is described in this section. There are four levels of priority in the table which we define in the following way:

Critical - This priority is given to areas which are absolutely necessary to obtain the desired write- and readability.

Important - This priority is given to areas which are important and should be considered thoroughly when taking decisions.

Less Important - This priority is given to areas which only have importance in some cases, and therefore can be of less priority.

Irrelevant - This priority is given to areas which has no effect on the write- or readability of the language.

Priority → **Criteria ↓**	Critical	Important	Less important	Irrelevant
Simplicity	✓			
Data Types and Structures			✓	
Abstraction		✓		
Generality				✓
Cost		✓		
Portability				✓

Table 2.1: Priority of language criteria.

Simplicity (Critical)

We define simplicity as the correct amount of basic components, or functionality, needed to make the language both read- and writable. Many basic components makes it hard for the programmer to learn all the functions of the language, and it also makes it hard to choose the right components at the right time. For instance in C/C++ there are quite a few ways to increase a variable (x++, ++x, x+=1, x=x+1), but on machine level these operations are not necessarily identical, and can sometimes have influence on the end result, so it can be difficult for the programmer to know which one to use [Kir02]. On the other hand a very restricted component group will make the language too simple, giving the programmer a hard time writing even simple functions, and therefore we prioritize this criteria as critical.

Data Types and Structures (Less important)

The amount of different data types and structures also influence the read- and write-ability of a language. If, for instance, there were no boolean types, a programmer would be forced to use another type in a way that it was not intended. In this case, using the number 0 for false, and all other values for true, would be a suitable solution. Again, too many types could easily confuse the programmer and force him or her to focus on perhaps very small differences between types.

Abstraction (Important)

Abstraction, in this case, can be seen as abstraction for subprograms and abstraction for data. Without abstraction for subprograms (i.e. functions or procedures), pieces of code would have to be replicated throughout the program and clutter up code, which would otherwise have a nice flow. Generally subprograms decrease readability a little, since there is a need to jump to the piece of code with the given subprogram in it and then back again; especially if the user does not designate the functions with understandable names. However the writeability of a language is greatly increased since there is no need to repeat identical code over and over again, so therefore we have rated this as important.

Citation *One way to keep a language simple is to add new features through function calls, and avoid custom operators and syntax. Looking up a feature in an index of function calls is much easier than paging through a language reference to find out what, for example, the "$" operator does.* *Falko Poiker, [Rea02, page 522]*

Generality (Irrelevant)

Generality can be defined as how well the language can be used to a wide variety of applications. For instance most special purpose languages (such as mIRC scripting [Mb04, Section 7]) has a narrow application group, while languages, such as C++ or Java, can be used for a large amount of problems. Since this language is designed for the game GamNa, it is a special purpose language and therefore we do not give this high priority.

Cost (Important)

The cost of a programming language is dependent on a subset of cost criteria, which in the end creates the final cost. Here cost refers to the time it takes in specific areas when creating a bot:

Program Creation The time is takes to create a program and debug it, which is highly dependent on the ease of write-ability and readability. High level languages tends to have a lower program creation time than low level languages, due to ease of readability[Seb04, page 18].

Program Execution This cost refers to the time it takes to execute a given program, which is influenced by the design of the language. A language with large amounts of type checking during run time prohibits fast execution (but may reduce compile time).

Program Maintenance How easy it is to maintain the final program. This is highly dependent on the readability of the language. Poor readability makes it harder for the programmer to maintain the program if he or she was not one of the original creators[Seb04, page 20].

Since this language is intended to run on an automated game engine, with no interaction from users, the program execution time is nearly irrelevant. Since it is a goal to create bots easily in the language, there should be a high priority to program creation and maintenance, which will greatly influence write-ability and readability.

Portability (Irrelevant)

How easy is it to port the final program to another platform, or another type of hardware. Portability is ensured by good standards, which unfortunately is a difficult and long process [Seb04, page 20][Rea02, pages 541-554]. There is no need for portability since LagNa is created to run on the GamNa engine, and the engine only works on Java compatible platforms.

2.2 The Language

In this section we will give a more thorough explanation of the language, including which aspects the language should support and why they has been chosen. Furthermore it will also be described which program paradigme we have choosen to use.

2.2.1 Programming Paradigm

This section is written with inspiration from [Seb04]. Before chosing which functions should be supported in LagNa, it is necessary to define which program paradigme LagNa should be used. The reason this is necessary to define is because different paradigme have needs different kind of functionality. The three most common programming paradigms are functional, object-oriented and imperative. The functional programming paradigm is used to mimic mathematical functions as precisely as possible. This is done to free the programmer from the troubles of the underlying architecture, and instead let him think as he would do if it was a mathematical problem. Also functional programming languages has the ability to return functions, which is a thing not seen in other paradigms. Object-oriented languages uses objects as an abstraction for data types, which allows for collections of commands and data. These objects can inherit characteristica from other objects, which is done to make reuse of code easy. Imperative programming languages are constructed to match the architecture of the underlying machine, resulting in good perfomance. Due to the good perfomance this style of paradigm is widely used.

We have decided to use an imperative paradigm for our language, because it is simple to implement and understand. Also it is very likely that a programmer who would use our language has tried other imperative languages. Even though we have choosen to create an imperative language, it is still somewhat special since the game is event driven.

2.2.2 Design Criteria

The first and most important design criteria is that our language should support all aspects described in Table 1.1 on page 5. If some of the functions in the game were not supported, it would not be possible to implement all features from the game into a bot. Therefore, it would not be possible for a bot to utilize its full potential in the game, and better bots could be developed in Java (which is the native language of the engine), using direct calls to the API of the engine. Furthermore we have decided to support constructs in our language which are not to be found in the API of the game, such as loops, conditions, mathematics functions and random generator. These constructs will make the language more easy to write programs in. The functionality which we have decided to implement in our language is based on our knowledge and experience with other programming languages.

We have split the programming functionality in two parts: one for the control of the programming flow, and another for user-friendly functions.

Controlling Program Flow

The program must be able to manipulate the bot. Like all programming languages, it must have a set of fundamental features to control the flow of the program using operators, statements etc. Below is a list of necessary features which, in our opinion, cannot be left out of an imperitive programming language as LagNa:

Variables The use of variables will be necessary in our language because conditions in the game may vary. For instance when calculating, an intermediate result it is often necessary to complete the calculation. A variable might be necessary to hold the intermediate result until the result is needed in the calculation again.

Conditions Sometimes it is required to make a choice, which will not be possible without conditions. We model this with an "if" construct. For instance, a bot might need to choose either to shoot or run away depending on the circumstances involved. These circumstances could for example be health status, weapon status or some other variables.

Loops In our language we need to have the possibility of using loops. Without loops, it would be required to either construct the piece of code as a recursive function or to state every command in sequence, which will make the program very large and confusing. Especially if the number of times a piece of code needs to be executed will vary. We model this with the "while" construct.

Mathematics To perform calculation it will be crucial to have some mathematics routines to use. For this we have decided to implement addition (+), subtraction (-), multiplication (*) and division (/).

Lists The possibility of using containers in a programming language can be highly useful. Such a structure makes it easy to store a large number of values, without having to assign them to individual variables. We have chosen to model a list with indexes to identify each value from each other.

Functions When programming, it is often valuable to be able to reuse some parts of already written code by the use of functions. It will be possible to use the code multiple times without writing the code again, but only by calling the function the code lies in. For instance, in the C code below (Listing 2.1), we could define a function to calculate the factorial of an integer number. This makes it possible to reuse the same code which improves writeabillity, and in most cases also makes the code easier to read since the code can be structured more logically.

Userfriendly Functionality

We have chosen to support additional language constructs to make the language more user friendly, and thereby ease up the process of development. The functionality of these constructs could have been obtained by the user with the use of all the traditional constructs, but instead of letting the users

```
int Factorial(int number)
{
  if number <= 1
  {
    return 1;
  }
  else
  {
    return number * Factorial(number - 1);
  }
}
```

Listing 2.1: An example of a factorial function using C

write large pieces of code every time they need them, we have chosen to do some of the time consuming work in advance. These constructs are:

Case Condition We have chosen to model a case construct in our language in order of making the check for multiple cases simpler, and make it easier to obtain an overview of the control flow. A case condition could look like Listing 2.2

```
Case x {
  1: DoSomeThing(1)
  2: DoSomeThing(2)
  3,4-6: DoSomeThing(3)
}
```

Listing 2.2: A pseudo code for the case condition statements.

This case statement might as well be constructed using the if-construct like this:

```
if x = 1 DoSomeThing(1)
if x = 2 DoSomeThing(2)
if x = 3 or x>= 4 and x <= 6 DoSomeThing(3)
```

Listing 2.3: Replacement of the case condition with if constructs.

It is therefore not a highly important construct, but nevertheless may prove useful in conditions where a value must be hold up against many other values.

Mathematical Functions Since we have prioritised simplicity as critical (see Table 2.1 on page 8), and simple mathematical functions can be useful when programming bots, we have decided to support some of the mostly used ones our language: The math functions we have decided to implement is:

- Random
- Square root
- Power
- Sine
- Cosine

Most of these functions are normal mathematical functions. Square root gives the square root of a given input, same behaviour with sine and cosine, however their output will be sine and cosine to the value of the input. However power is a little different then normally, where for instance x to the power of two is written as “xˆ2”, however in our language this should be written as “power(x,2)”. The reason “ˆ” is not used is because we have choosen to implement all the mathematical functions as function calls, to make it easier for new mathematics be added later on. If we chose to update the language in this way, it would be much more easy for the programmers to learn new names of function calls than to learn new tokens for the language. [Rea02][page 522]. This can also be observed from other languages: for instance, every time Java comes in a new realease, new functions have been made; however new tokens is not added very often. [java] show the all updates made in J2SETM 1.4.2 Beta. Random is, in our language, a function which, given a minimum and a maximum value, will return an integer between the minimum and maximum value. For instance random(5,10) would return a random number in the interval [5;10].

2.2.3 Scopes

This section have been inspired from [WB99][page 137-144] When dealing with a programming language which support declaring of variables, functions or something similarly, it is necessary to define which scope rules apply. Scope rules define which declared variables or methods a construct knows about at a given time. A scope can be described as a block of commands; if a variable is declared inside this block, all subsequent commands in the same block can “see” and access this variable, but commands outside the block cannot “see” what is inside the block and therefore cannot access any variables declared inside the block. Different kinds of scope rules can be used for describing the behaviour of a language. The three scope rule types described below is displayed in Figure 2.1 on the next page.

Monolithic This kind of scope rule allow only one block for all variables to be declared in. This means that all variables is declared as global variables. Therfore, everything has access to all previous declarations in a program, using monolithic scope rules.

Flat This kind of scope rule allow more than one block of commands to be used in a program, but no overlap or nesting is allowed. This could for instance be a new scope for every function in a program.

Nested With nested scope rules it is possible to make scope blocks inside of other scope blocks. This means that e.g. a function declared in scope A can have a scope inside itself called B, and B can have a scope inside itself called C. Code inside C can access everything in B and A, and code inside B can access everything in A. But A cannot access anything inside B or C, and likewise for for B, which cannot access anything inside C.

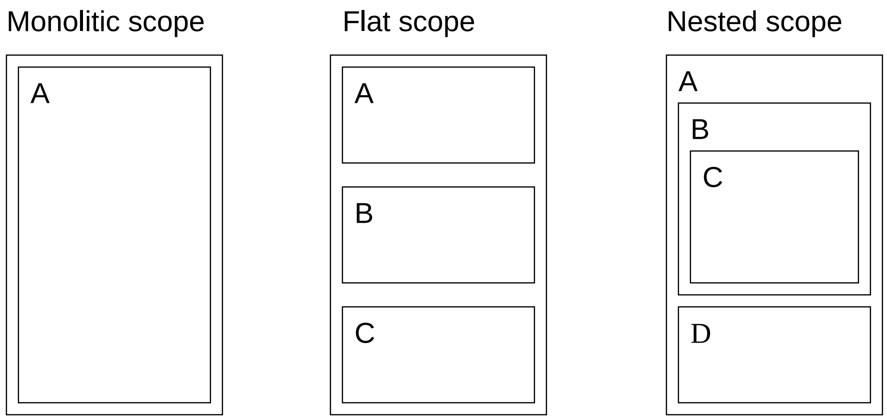


Figure 2.1: The three scope types.

Normaly monolithic scope is not used for programming languages designed for writing huge programs, normaly because everything have access to all variables in the program. This would mean that different employees working together on the same piece of software, should watch out for re-declarering variables, and by using the same variables, could easily leed to errors. Also, normaly a huge program uses lots of different variables, but by using monolithic scope rules all variables would have to be loaded into the memory at the same time, which could leed to an unnecessary huge memory allocation.

In flat scope rules it is possible to declare variables with the same name, as long as they do not get declarered more than once in a scope block. With flat scope the memory consumption of the program will be smaller, as not all scopes have to be loaded from the start but only when they are to be executed.

Nested scope rules allow for variables declared in outer scopes to be used in inner scopes. Nested scopes, like monolitic scopes, allows for global variables where flat scope does not. There is a problem with naming in nested scope. Say a variable is declared inside B and the same variable is declared inside A. Which should then be used inside B? Normaly the variable last declared when the scope was declared is used (static binding) which in this case would be the variable declared inside B. An alternative to static binding is dynamic binding where the variables which was last loaded into the memory is used. This makes it hard to know if the variable declared in A or B is used.

To illustrate the difference between static and dynamic binding, we have constructed an example shown on Listing 2.4.

```
1    Integer MaxValue = 50;
2
3  {
4    integer i = 100;
5    string IsLess = "";
6
7    if(IsLessThanMaxValue(i) == false)
8    {
9      integer i = 2;
10     IsLess = "not";
11   }
12   print(i + " is " + IsLess + " less than " + MaxValue);
13 }
14
15 function boolean IsLessThanMaxValue(integer x)
16 {
17   if(x < Maxvalue) return true;
18   else return false;
19 }
```

Listing 2.4: A Pseudo code example which show the different between static and dynamic binding.

If this code example is using dynamic binding, line 12 would use the variable “i” declared in line 9 because it is the variable last loaded into the memory with the name “i”. The output from this program would be “2 is not less than 50”.

If this code example instead of using dynamic binding would use static binding, line 12 would not know anything about variables declared indside the “if scope” (line 7 to 11). The output from this program would be “100 is not less than 50”.

We have choosen to use nested scope rules and static binding. The reason we have choosen to use nested scope rules is because it makes it possible to declare temporary variables for the only purpose of existing inside an if or while block. This also make it possible to declare variables close to where they are going to be used. However we have chosen not to support forced scopes. This means that it will not be possible in LagNa to force a new scope with use of a start scope and a end scope, it is only possible to start a new scope by use of constructs in LagNa, these construct will declare a new scope when used, the construct is: while, if, case and methode construct.

For variable binding we have choosen to use static binding for LagNa because it eases up the readabillity and the writeabillity of code written in LagNa. This is because when using static binding, commands can only "see" variables declared inside their own scope and all variables declared in earlier scopes, but not inside subsequent scopes. This mean that it is possible to take a look at the code and thereafter know which variables can "see" each other. In contrast when using dynamic binding, it is necessary to run through the code before knowing which variables is going to be used at a specific place in the code. For instance if "IsLessThanMaxValue" in Listing 2.4 had returned true instead of false, the variable "i" would not have been redeclared and therefore the declaretion at line 4 would be used in the "print" command at line 12. Therefore, different declarations of the same variable name can be used by the same command, which could lead to confusion.

Summary

In this chapter all our quality design have been described, these will be used to priority our work through the rest of the report This chapter have also described which construct LagNa should support. To continue in the process of develop a language which later can be implemented,so a compiler can be made, it is necessary to declare a syntax for LagNa, this process will be described in the next chapter.

Chapter 3

Syntax

In the previous chapters, we have examined what our programming language must be able to do. The next step in the development phase is to describe the syntactic properties of the language. That is, how the program is actually written.

We will introduce, a precise way of describing those syntactical properties, and while using that, we will go through each of the individual areas of the language one by one.

3.1 Context-Free Grammar

A *context-free grammer*(CFG), is a formal way of describing languages, by specifying how any text in that language can be produced, by following a set of substitution rules from a start variable. The term *context-free* comes from the fact that any substitution rule can be used independently of its context.

The following definition for Context-Free Grammers, is written with inspiration from [Sip96].

Definition 3.1. Context-Free Grammar
A CFG is a 4-tuple (V, Σ, R, S), where

- V, is a finite set of **variables**.
- Σ, is a finite set of **terminals**, where $\Sigma \cap V \equiv \emptyset$.
- R, is a finite set of **rules**. A rule consists of a variable, and a string of variables and terminals.
- S, is the **start variable**, where $S \in V$.

In Example 3.1 we show a simple CFG using Definition 3.1. It gives a CFG for the mathematics of an unknown variable and the operations '+' and '$\cdot$'.

Example 3.1.
We define a grammar

$$G = (\{<EXP>\}, \{x\}, \{R\}, \{<EXP>\}),$$

where R is

$$<EXP> \rightarrow <EXP>+<EXP> \mid <EXP> \cdot <EXP> \mid x$$

◆

The sequence of substitutions, to generate a string, in a language is called a *derivation*. A derivation, in which at every step, it is the leftmost remaining variable, which is substituted, is called a *leftmost derivation*. Equivalently we have rightmost derivations. A grafical way of illustrating *derivations*, are via derivation trees. Example 3.2 show such a tree.

Example 3.2.
A derivation of the string $x+x$, which are composited from the grammar shown in Exampel 3.1, would be:

$$<EXP> \Rightarrow <EXP> + <EXP> \Rightarrow x + <EXP> \Rightarrow x + x.$$

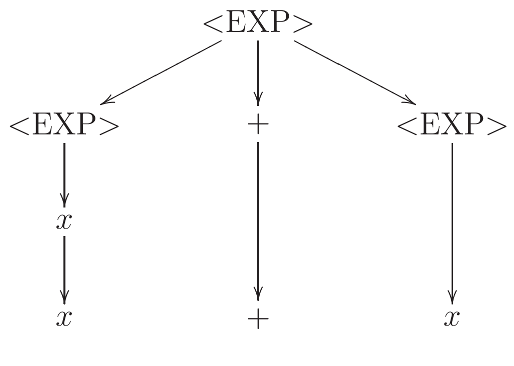


Figure 3.1: Derivation tree for the string "$x+x$" using the grammar from Example 3.1

◆

We now use leftmost derivations in a definition of ambiguity in grammars.

Definition 3.2. Ambiguity of Grammars
We say that a string is derived **ambiguously** in a Context-Free Grammar G, if it has two or more leftmost derivations.

We say that G is **ambiguous**, if it generates some string ambiguously.

We return to Example 3.1 on the preceding page and show that it is ambiguous.

Example 3.3.
We will show that the grammar G from Example 3.1 generates the string $S = x + x * x$ *ambiguously, by giving two different leftmost derivations. See Figure 3.2 for the two derivations.*

◆

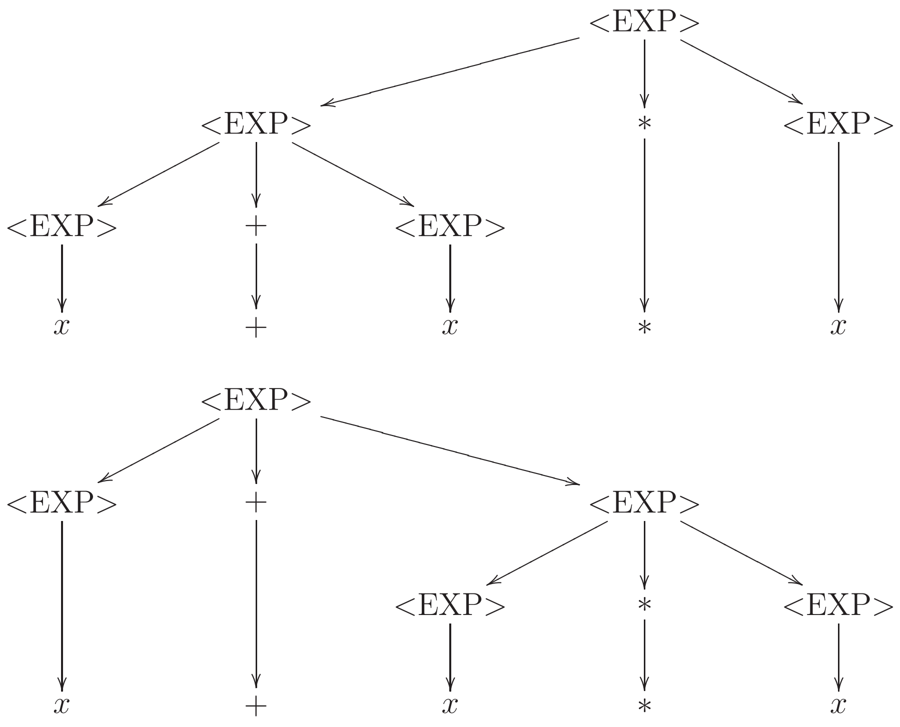


Figure 3.2: Derivation trees for the string $x + x * x$ using Example 3.1

When making a language, it is important that all ambiguities are removed, since a compiler would have to choose, which it is not able to.

There are several different notational ways of presenting a grammar. In this report we have chosen to use the Extended Backus-Naur Form, following the International standard *ISO 14977*, [ISO] (a final draft is located at [fS96]). The most important notations to keep in mind when reading our syntax is the following:

- All terminal symbols are quoted using ‘ and ’.
- [and] indicates that the enclosed symbols are optional
- **{** and **}** means repetition (should be followed by the repetition-symbol)
- **(** and **)** are used to group items together
- The operators are (ordered in highest precedence first):

  1. The repetition-symbol $*$, repeates the group zero or more times
  2. The except-symbol $-$
  3. The definition-separator-symbol |
  4. The defining-symbol $=$

Again we return to our previous example and show an unambiguous version of that grammar in our new notation.

Example 3.4.
We redefine the grammer to

$$G = (\{EXP\}, \{x\}, \{R\}, \{EXP\}),$$

where R is

$$EXP = \text{‘}x\text{’}\ [\ (\ \text{‘}+\text{’}\ |\ \text{‘}*\text{’}\ )\ EXP\ ]$$

◆

The grammer in Example 3.4 forces the compiler to match the first terminal, immediately before continuing, instead of the previous grammar, that allowed the compiler to match different parts first. Therefore, this grammar does not have the ambiguity problems, as the original grammar. See Figure 3.3 for the derivation tree.

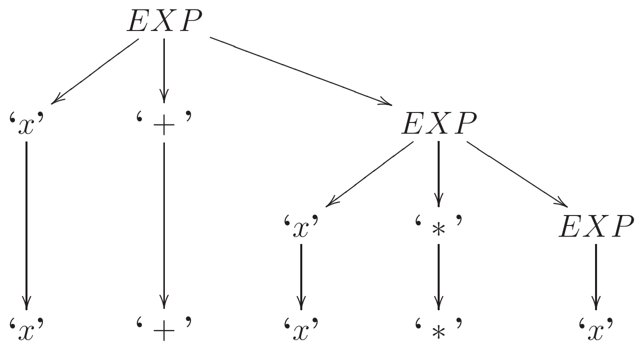


Figure 3.3: Derivation tree for the string $x + x * x$ using Example 3.4

3.2 Overall Structure

A bot description can be divided into 4 elements, which makes up the entire bot. These four elements are shown in Table 3.1. The four elements are: global variable type declarations, an entry point named "run", events and methods. Programmers are required to declare all global variables before all other, then the events, methods and "run" can be declared. They can however intertwine events, methods and "run".

The four tuple for our language is:

$$G = (V, \Sigma, R, S)$$

- V: Our variables can be seen in Appendix B.
- Σ: Our terminals can be seen in Appendix C.
- R: All the rules are defined in the next sections.
- S: The start variable is *bot*.

bot	=	{ G-type-decl }* { bot-element }* [run { bot-element }*]
run	=	'run' '(' ')' '{' commands '}'
bot-element	=	event | method

Table 3.1: Main elements

Specific type declarations will be described later in Section 3.3 on the next page. Here they are used globally in the sense that they are accessible from all parts of the program.

The run method will be called whenever the bot has finished running through any of the called events. This will be described in Section 3.5 on page 29.

Methods will be discussed in Section 3.6 on page 30 and events in Section 3.5 on page 29.

3.3 Type Declaration

Type declarations are used for declaration of new variables in the code. A variable in our language is the same as a variable in mathematics. We have choosen not to allow special characters in our variable names. The reason for this is that we want to avoid, that a variable name can be interpreted as something else, fx. if one declare a vaiable as *r()*, it could be mistaken for a call of the method named *r*. The syntactical rules for variable names (*var-name*) can be seen in Table 3.2.

<table>
<tr><td>types</td><td>=</td><td>real</td></tr>
<tr><td></td><td>|</td><td>bool</td></tr>
<tr><td></td><td>|</td><td>string</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>var-name</td><td>=</td><td>var-char { var-char | digit }*</td></tr>
<tr><td>var-char</td><td>=</td><td>‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’</td></tr>
<tr><td></td><td>|</td><td>‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘y’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’</td></tr>
<tr><td></td><td>|</td><td>‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’</td></tr>
<tr><td></td><td>|</td><td>‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘Y’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’</td></tr>
<tr><td>number</td><td>=</td><td>digit { digit }*</td></tr>
<tr><td>digit</td><td>=</td><td>‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ | ‘0’</td></tr>
</table>

Table 3.2: Variables

Before a variable can be used, it needs to be declared. To declare a variable it is necessary to specify the type. In LagNa a variable can have different kind of types, see Section 3.4 on the facing page for the types. When a gobal variable are declared they can be intialized, but only to a value and not an expression. See Table 3.3 for the syntactical rules for declaring a variable.

When declaring variables, some means of assigning values to them, (initialization) is desirable. For this our language supports an assignment operator (see Table 3.4).

type-decl	=	identifier var-name [':=' exp] ';'
G-type-decl	=	identifier var-name [':=' types] ';'
identifier	=	id [list]
id	=	'real' | 'boolean' | 'string'
list	=	'[' ']'

Table 3.3: Type Declarations

assignment	=	var-name [ID-list-call] ':=' exp ';'

Table 3.4: Assignments

3.4 Data-types

There are three data types in our languange; real, boolean and string. We have choosen only these three data types because we believe it is possible to write bots, without the use of advanced data types. We have chosen not to have support for user defined types, which means that the user will not be able to combine the primitive types into an abstract type. The reason for this limitation of LagNa is that we do not consider this to be an essential feature, and therefore prioritize other parts of the language higher.

The syntax for the three different types, *real, boolean* and *string* will be described in Section 3.4.1, 3.4.2 and 3.4.3.

3.4.1 Real

A variable of the type real can be any real number in a specified range. This range is dependant on the implementation, so it can differ from platform to platform. This lowers the portability of the language, but is of no real concern since the engine handles the portability (see Portability on page 10).

We have chosen not to include an integer type, because the user can use the real as if it was an integer by omitting the comma, for instance 10.0 can be written as 10 if the programmer wishes (see Listing 3.1).

The reason why we have chosen reals, is due to the loss of precision in calculations, with the use of integers. An alternative could be to use both types, but this would compromise our quality goals of simplicity (see quality goals at Section 2.1 on page 7).

See Table 3.5 for the syntactial rules for reals.

```
//A  variable of type real, called A, assigned with the value
    10.0
real A := 10.0;
//Another variable of type real, called B, with the value 10
    assigned.
real B := 10;
//A variable of type real, called C with no value assigned.
real C;
```

Listing 3.1: The syntax for declaring a variable of type real

real	=	['-'] number ['.' number]

Table 3.5: Reals

3.4.2 Boolean

The type boolean can hold only two different values: either true or false (See Table 3.6 for the syntactial rule). Booleans are often used to control program flow through the use of control structures such as the if-statements (see control structures at Section 3.7.2 on page 32).

bool	=	'true' | 'false'

Table 3.6: Booleans

3.4.3 String

Variables of the type string can hold any UNICODE characters [Uni04]. Each string has to be declared with a surrounding pair of " which is not included in the string itself.

Although string is a data type it differs from the other data types in that it is not possible to do boolean operations on it. This means that it is not possible to test if two strings are equal. This limits the usability of the string type, but even though, it can also serve as a tool for debugging.

We have no single-character type because such a type can be modeled as a string of one character's length. Some special characters could be evaluated even though they are in the string, such as the " character; these can be expressed by using an escape character. We have chosen to use a backslash (\) as this escape character. This way we can write a string containing quotation marks as shown in Listing 3.2.

See Table 3.7 for the syntactical rules.

string	=	‘"’ { chars }* ‘"’
chars	=	char
	|	‘\"’
	|	‘\\’
char	=	uni-chars - (‘"’ | ‘\’)
uni-not-eol	=	uni-chars - (end of line)
uni-chars	=	UNICODE characters

Table 3.7: String

Variables in Strings

It is possible to include variables of different types in strings via concatenation using the sign &. Also in this way it is possible to combine strings into larger strings. See Listing 3.2 for an example on string syntax.

```
//A variable of type string, called A, assigned with the value
    Hello
string A := "Hello";
//Another variable of type string, called A with no value
    assigned.
string A;
//A variable of type string, including an escape character.
//Produces the output: Hello to "You"
string B := "Hello to \"You\"";
//A variable of type string, is concatenated with another string
//variable and a real variable giving a resulting string D.
real C := 100;
string D;
D := A & B & C; //Outputs: HelloHello To "You"100
```

Listing 3.2: The syntax for declaring a variable of type string

3.4.4 ID-list

The type ID-list can hold an arbitrary nummer of elements of a single data type. To assign a value to an entry in the list, a key has to be selected, so that the value can be retrived again. See Table 3.8 for the syntax of ID-list.

ID-list-call	=	‘[’ exp ‘]’

Table 3.8: ID-list

See Listing 3.3 for an example on how to assign a value to a list and retrive it again.

```
//The  value 5 is assigned to a ID-list called "list" with the key 1
list[1] := 5;
//The value is retrived from the list using the key
x := list[1];
```

Listing 3.3: The syntax for declaring and assigning a value to a list and retrive it again.

3.4.5 Operator Precedence

For operator precedence to be implemented in the syntax of a language, operators must be in different levels of a generated derivation tree. Operators in lower parts of the syntax tree has higher precedence than those in the top, as shown on Figure 3.4.

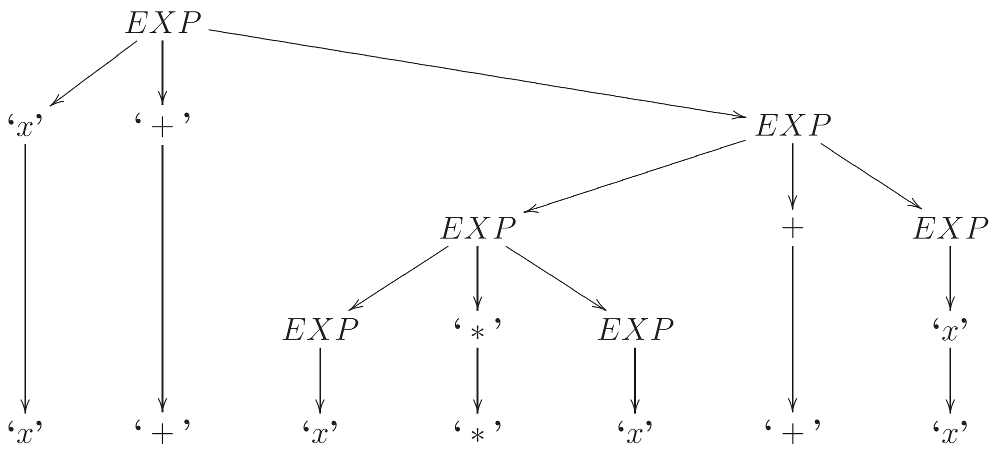


Figure 3.4: Derivation tree for the string $x + x * x + x$ to show precedence.

Example 3.5 shows the importance of precedence. It shows that different precedence can give different results.

Without precedence rules the computer has no way of knowing how to complete the calculation. In accordance with our quality goals of simplicity (see Table 2.1 on page 8), we want to use precedence rules, which are as identical to mathematical precedence as possible, i.e. multiplication and division signs precedes addition and subtraction.

Example 3.5.

$$
\begin{aligned}
1 + 2 \cdot 3 &= (1 + 2) \cdot 3 = 3 \cdot 3 = 9 \qquad \textit{Non-mathematical precedence} \\
& \textit{or} \\
1 + 2 \cdot 3 &= 1 + (2 \cdot 3) = 1 + 6 = 7 \qquad \textit{Correct precedence}
\end{aligned}
$$

◆

For the reals we have 3 sets of operators, with different precedence. The first two is the mathematical operators. And as seen in Example 3.5 the multiplication ($*$) and division ($/$) signs has higher precedence than, addition ($+$) and subtraction ($-$). The next operators, of lower precedence, is the comparison operators ($<, >, <=$ and $>=$), which gives a boolean as result. The next precedence level holds the last operators, which works on reals. These are the equality operators: equal ($=$) and not equal ($<>$). Although they can also work on booleans, but with the result being a boolean in each case.

The operators of even lower precedence, which only work on boolean expressions, are the boolean operators (AND, OR and NOT). Here we use the same idea for precedence rules, that we used for operators on reals (see Example 3.5).

The only operator that works on strings is the concatenation (&), which has the lowest precedence of all operators.

To make sure that the expressions are not ambiguous, we have put all the different expressions into one variable. So when one writes *exp* it can be all the different expressions, that are possible in LagNa. The way it works is, that all the expression-variables call the next expression-variable and/or some operator and itself.

See Table 3.9 for our syntactical precedence rules and Appendix D for a derivation tree over the string "2+2".

3.5 Events

Events are as mentioned important, since they are the means the engine uses to communicate with the bots about any events that might have occurred in the game, which are relevant to each bot. Events are where the programmer can control the bot's behavior in response to the game. Events can be seen

exp	=	bool-exp [‘&’ exp]
bool-exp	=	bool-exp-1 [(‘AND’ | ‘OR’) bool-exp]
bool-exp-1	=	[‘NOT’] real-exp
real-exp	=	real-exp-1 [(‘=’ | ‘<>’) real-exp]
real-exp-1	=	real-exp-2 [(‘<’ | ‘>’ | ‘<=’ | ‘>=’) real-exp-1]
real-exp-2	=	real-exp-3 [(‘+’ | ‘-’) real-exp-2]
real-exp-3	=	value [(‘*’ | ‘/’) real-exp-3]
value	=	[‘-’] (‘(’ exp ‘)’ | var | special)
	|	types
var	=	var-name [method-call | ID-list-call]

Table 3.9: Precedence

as predefined methods that the programmer cannot invoke directly. The programmer is not required to declare or implement all the events. He can decide only to declare and implement the events the bot should react to. The engine will automatically, ignore all events not declared explicitly by the programmer.

See Table 3.10 for the syntactical rules for all the events.

event	=	‘event’ event-name ‘{’ commands ‘}’
event-name	=	‘OnHit’ ‘(’ ‘real’ var-name ‘)’
	|	‘OnDeath’ ‘(’ ‘string’ var-name ‘)’
	|	‘OnCollision’ ‘(’ ‘real’ var-name ‘,’ ‘real’ var-name ‘)’
	|	‘OnSight’ ‘(’ ‘real’ var-name ‘,’ ‘boolean’ var-name ‘,’ ‘real’ var-name ‘)’
	|	‘OnTargetHit’ ‘(’ ‘)’
	|	‘OnStart’ ‘(’ ‘)’
event-list	=	event-list-name { ‘,’ event-list-name }*
event-list-name	=	‘OnHit’ | ‘OnDeath’ | ‘OnCollision’
	|	‘OnSight’ | ‘OnTargetHit’ | ‘all’

Table 3.10: Events

3.6 Method

Methods are a combination of functions and procedures (they are in no way related to methods in object oriented languages, where they operate on objects). When there exists a return-type, the method works like a function, and when none is defined, the method works like a procedure. If the method has a return-type, the return command needs to be specified with an expression of the same type as the return-type. The ”return” token, if present, must be the last command in the method. Also, every method name has to be unique.

See Table 3.11 for the syntactical rules for methods.

method	=	'method' method-tail
method-tail	=	return-type var-name '(' [parameters] ')' '{' commands 'return' exp ';' '}'
	|	var-name '(' [parameters] ')' '{' commands '}')
return-type	=	identifier
parameters	=	parameter {',' parameter }*
parameter	=	identifier var-name
method-call	=	'(' [method-inputs] ')'
method-inputs	=	exp {',' exp }*

Table 3.11: Method

3.7 Actions and Control Structures

The only code a programmer can write in an event or method is a sequence of commands. Commands are almost all the other rules assembled in one rule, expect event and method declarations. Comments can be written anywere in the code, because the rest of the line will just be ignored by the compiler when it reads the code. See Table 3.12 for the syntactical rules for commands and comments.

commands	=	{ command }*
command	=	action
	|	if-exp
	|	while-exp
	|	case-exp
	|	assignment
	|	method-call ';'
	|	type-decl
comment	=	'//' uni-not-eol

Table 3.12: Commands

3.7.1 Actions

Actions are significant commands for the bot. This is the only way for the programmer to tell the engine what he wants the bot to do. This is where LagNa implements all the features from GamNa. See Table 3.13 for the syntactical rules for actions.

action	=	action-name ‘)’ ‘;’
action-name	=	‘ignore’ ‘(’ event-list
	|	‘unignore’ ‘(’ event-list
	|	‘rotate’ ‘(’ exp
	|	‘rotateleft’ ‘(’
	|	‘rotateright’ ‘(’
	|	‘look’ ‘(’ exp
	|	‘lookleft’ ‘(’
	|	‘lookright’ ‘(’
	|	‘strafe’ ‘(’ exp
	|	‘strafeleft’ ‘(’
	|	‘straferight’ ‘(’
	|	‘move’ ‘(’ exp
	|	‘moveforward’ ‘(’
	|	‘movebackward’ ‘(’
	|	‘chooseweapon’ ‘(’ exp
	|	‘shoot’ ‘(’ [exp]
	|	‘say’ ‘(’ exp
	|	‘stop’ ‘(’
	|	‘stopmove’ ‘(’
	|	‘stoprotate’ ‘(’
	|	‘stoplook’ ‘(’
	|	‘stopshoot’ ‘(’
	|	‘waitforevent’ ‘(’
	|	‘waitforall’ ‘(’
	|	‘waitformove’ ‘(’
	|	‘waitforrotate’ ‘(’
	|	‘waitforlook’ ‘(’
	|	‘waitforshoot’ ‘(’

Table 3.13: Actions

3.7.2 Control Structures

It should be possible to let the bot make decisions based on some sort of data input. For this, two conditional constructs are supported in the language. These are the *if-else* structure and the *select case* structure. See Table 3.14 for the syntactical rules of the if-else construction.

if-exp	=	‘if’ exp ‘{’ commands ‘}’ [‘else’ ‘{’ commands ‘}’]

Table 3.14: If else

We have chosen to remodel the switch construct from the one used by Java or C. A description of the switch statement in Java can be found at [Swi]. We have designed the select construct, so that each case is followed by an implicit break. This way we avoid running all cases followed by the case which was matched. Additional features to our select construct includes the ability to choose intervals. This is done using almost the same notation as in the mathematical world. The syntactical structure can be found at Table 3.15. See Listing 3.4 for an example of a select statement.

select-exp	=	‘select’ exp ‘{’ cases ‘}’
cases	=	{ case }* [default]
case	=	case-exp ‘{’ commands ‘}’
default	=	‘default’ ‘{’ commands ‘}’
case-exp	=	case-exp-1 { ‘,’ case-exp-1 }*
case-exp-1	=	(‘]’ | ‘[’) real ‘:’ real (‘]’ | ‘[’)
	|	real

Table 3.15: Select

```
real test := 3;
select (test)
{
   1 { say(" The variable is 1"); }
   [2:4] { say("The variable is between 2 and 4"); }
   ]4:8[ { say("The variable is between 4 and 8"); }
   9, 10, 11 { say("The variable is either, 9, 10 or 11"); }
   [12:15], 20 { say("The variable is either between 12 and 15, or 20"); }
}
```

Listing 3.4: Select statements

3.7.3 Loop Structures

To make repetition easy, we have included a loop structure called *while*. *While* functions like in other languages such as C and Java, in that it takes a boolean operation, and executes the enclosed code, if the boolean expression evaluates to true. The syntactical rule of the while stucture can be seen at Table 3.16.

while-exp	=	‘while’ exp ‘{’ commands ‘}’

Table 3.16: While

3.7.4 Special Functions

There are some functions in LagNa which we have made, to make it easier for the programmer, to make, eq. geomatric calculations, or to get information about the bot. See Table 3.17 for the syntactical rules.

special	=	'sin' '(' exp ')'
	|	'cos' '(' exp ')'
	|	'power' '(' exp ',' exp ')'
	|	'sqrt' '(' exp ')'
	|	'random' '(' exp ',' exp ')'
	|	'gethealth' '(' ')'
	|	'getviewingangle' '(' ')'
	|	'canshoot' '(' ')'

Table 3.17: Special-functions

3.8 Summary

In this chapter we have defined the syntactical rules for LagNa. For an example of our syntax in use, see Appendix G. Now it is time to describe the sematics of LagNa.

Chapter 4

Formal Semantics

In the previous chapter we looked at the grammatical structure, or syntax, of the LagNa language. Semantics, on the other hand, is concerned with the behaviour of grammatically correct programs. In this chapter we will explore this behaviour using formalised mathematical notation. It has been written with inspiration mainly from [Hüt04].

4.1 Approaches in Semantics

Several different ways of denoting the behaviour of languages have been proposed over the years. The most commonly used are:

Operational semantics describes an execution on an abstract machine, using registers, stacks etc. Operational semantics describes how the effect is achieved in terms of the computation steps it takes in an idealized environment [Hüt04, Chapter 3].

Denotational semantics uses functions in a mathematical domain to describe only the effect - not how it is obtained [Hüt04, Chapter 13].

Axiomatic or logical, semantics describes specific properties of the effect by setting up assertions for each constructs in a program. This models the relationship between pre- and post-conditions of program variables [NN99, Page 2].

These different types of approaches are suitable for describing different aspects of a program. Since we have devoted a part of this project to the implementation of a compiler for LagNa (see Chapter 6), we have chosen operational semantics to describe the language. This way we define the meaning of each construct by giving an abstraction to how the program is executed in the machine, which will make it easier to understand the actual implementation. Later in Chapter 5 on page 97, we will also look at axiomatic semantics as a tool for verification of program properties.

Operational semantics comes in two flavours:

Big-step, or structural, semantics describes how the overall results of executions are obtained

Small-step, or natural, semantics describes how the individual steps of the computation takes place

Small-step is a newer approach to operational semantics which have certain advantages compared to big-step, especially when it comes to modelling parallelism. The configurations for small-step are the same as for big-step, and the two semantics are equivalent in the sense that all big-step semantics can also be modelled in small-step.

Before making the choice between small-step or big-step semantic it is necessary to discuss whether or not LagNa is a parallel language or not.

Sequentially or Parallel

In this section it will be discussed whether or not big-step semantics can be used to describe LagNa. The reason this discussion is relevant is because big-step semantics can not describe parallel performance. However there are both reasons that LagNa could be a parallel language, and reasons for that LagNa could be a sequentially language.

One of the arguments for that LagNa must be parallel is that it is possible to have more than one bot in the arena at the same time, and therefore it must be parallel. This could be compared to a C program which uses threads for parallel performance. In this case we compare the GamNa engine with the main C program and the bots written in LagNa could be compared to the threads spawned by the main C program. However instead of comparing GamNa with a normal C program, we could compare GamNa with an operating system. Each bot written in LagNa would then be a process running in the operating system GamNa. This comparison would still mean that GamNa is considered parallel, however now LagNa does not need to be parallel. It could still be parallel, because it is possible to write parallel applications for a parallel operating system. However sequentially applications can also be written to a parallel operating system, and still be executed just as fine as a parallel application could. An example of this could be the Batch

Script for DOS (Disk Operating System) which is a language designed to unite and execute commands from the operating system MS-DOS. Because MS-DOS is a sequentially operating system it is not possible to execute parallel applications and therefor Batch scripting for DOS is nether a parallel script language. However since the time of DOS, Microsoft has developed new operating system such as Microsoft Windows which is parallel, but even though Microsoft Windows is a parallel operating system, it is still possible to run Batch script written in DOS. Microsoft Windows can even run several Batch scripts at the same time. Therefor even though a language or a process is sequentially it can still run in a parallel operating system.

Another parallel characteristic which LagNa has, is the ability to react to events, this gives the bot's the ability to react to other bot's. This would make LagNa a parallel language because two different applications (bot's), running on the same computer, communicating with each other would mean parallel performance. But in GamNa bot's does not directly communicate with each other. The only communication is between a bot and the GamNa engine, and reversed between the GamNa engine and a bot, as described in section 1.2. This means that bots written in LagNa actual never sees another bot, it is only sees the GamNa engine which tell the bot that right now another bot is right in front of it. Another argument for LagNa to be parallel is that it receives information, from outside the language, (LagNa is an event based languages). But this doe not necessarily means that LagNa have to be parallel. An example for this could for instance be Crontab which is an application used in for Unix (see [Vix] for further information on Crontab). The idea with Crontab is that it can execute a specific Cron script at a specific chosen time of the day or month. Cron is a small sequentially script language much like Batch where it is possible to execute different Unix commands. Because Crontab can execute a script at a specific time it means that crontab is depending on external influence. Because if the computer which crontab is running on does not update the system clock in the computer frequently it will result in the Cron job not being executed, because "no one" gave Crontab the information it was dependent on. Another small example of outside dependence could be made up with use of the MS-DOS scripting language, and for instance Microsoft Word. Microsoft Word has a built-in autosave which saves the open document in a specific file, lets say MS-Word saves the autosave file in "c:\autosave.doc" However if a user of MS-Word wanted to delete this file every time MS-Word created this file it would be possible to achieve with the use of a small sequentially Batch script. The script could look like the code in Listing 4.1.

This small code looks for the file "c:\autosave.doc" and every time it finds the file it will delete it and start looking again.

```
@echo off
:Try–again
cls
if not exist c:\autosave.doc (
   echo File not found, looking again...
   goto Try–again
 ) else (
   echo File found!
   del c:\autosave.doc
   goto Try–again
 )
)
```

Listing 4.1: An example of a Batch script which is dependent on external influence

In LagNa it is possible to make a bot both move and rotate at the same time, for instance if a bot were to move around in a circle, shown in the code in Listing 4.2, it would be possible to describe this only by, calling two actions in LagNa, This will mean that both of the commands from LagNa are execute at the same time, and therefore LagNa would have to be parallel to maintain this kind of behavior.

```
run()
{
  move(360);
  rotate(360);
}
```

Listing 4.2: An example of a bot moving around in a circle

As seen in the semantic 4.36 on page 80 for "move" and 4.35 on page 79 for "rotate" it is shown that these two expression actual is evaluated sequentially. This means that a bot does not actually move it self. It only tells the GamNa engine which of the direction it wants to go (this is accomplished by the use of the provided actions in LagNa), and then the engine then decides how this is carried out.

A reason why LagNa is sequentially language is that there are no commands, action or functionality which support parallelism directly. An example of such a command could be a command named "par{command}{command}" which would take two commands as input and run these at the same time. For instance:

The different opinions, about LagNa being a sequentially language or a parallel language, discussed in this section can all be seen in Table 4.1

```
real x=0;

run()
{
  Par
  {
    real stop=0;

    While(stop < 5)
    {
      x := 2;
      x := x + 3;
      stop := stop + 1;
    }
  }
  {
    real stop=10;

    While(5 > stop)
    {
      x := x - 1;
      stop := stop - 1;
    }
  }
}
```

Listing 4.3: An example of how parallelism could have existed in LagNa

Sequentially	Parallel
Even though a system is parallel, processes running on the system can still be Sequentially	2 or more bots can be executed in the same arena at the same time
Sequentially applications can be depending on external influence, which means they are not alone	LagNa uses event call, therefor more than one bot have to run at the same time
A bot run every command or action sequentially	It seams that a bot written in LagNa can run than one command at the same time
2 Bots written in LagNa can not see each other, the GamNa engine tells the bots everything they need to know.	Because some event can only occur when 2 or more bots are running on the same arena at the same time they can communicate with each other, with use of event.
There exist no construct in LagNa which support parallel performance	

Table 4.1: Transition rules for big-step while statements

As seen in this small section both reasons for that LagNa is sequentially or parallel exist, and there is probably a lot of other reasons that LagNa must be one of these instead of the other. However to be on the safe side we have chosen to describe LagNa with use of small-step semantic. This way it will also be possible to make updates of the language LagNa easy, because if parallel constructions later on is wanted in LagNa, this could easily be implemented, because then all other constructs in the language would also be described with smallstep, which means that the wanted parallel performance could be made.

4.2 Modelling the Abstract Machine

In order of keeping the semantics more clear, we will introduce a set of *meta variables* which range over specific syntactic categories, as found in Chapter 3. We obtain instances of a rule by replacing the meta variables with concrete expressions from the given syntactic category. For the rest of this chapter, we will thus use the notation as seen in Table 4.2. Note that values also contain a special event value called EV. This will be explained in Section 4.4.

Sets	
$\mathbb{B} = \{tt, ff\}$	the set of booleans
$\mathbb{N} = \{0, 1, ...\}$	the set of natural numbers
$\mathbb{R} = \{...\}$	the set of rational numbers
$\mathbb{S} = \{...\}$	the set of strings
$\overrightarrow{x}$	the set of x, so $\overrightarrow{x} = \{x_1, ..., x_n\}$
$A \times B$	The cartesian product of A and B
A^*	$A \times A \times ... \times A$ * times.
Meta variables	
$e \in$ exp	arithmetic, string or boolean expression
$C \in$ commands	command or action
$l \in \mathbb{N}$	location
$x, y, M \in$ *var-name*	variable name or method name
$v \in \mathbb{B} \cup \mathbb{R} \cup \mathbb{S} \cup \{EV\}$	value of any type
Notation	
$\mapsto$	maps to
$\rightharpoonup$	partiel function

Table 4.2: Syntactic categories

4.2.1 The Environment-store Model

We will use an environment-store model to represent bindings of variables in different scopes. The model is an abstraction of how the bindings takes place in a computer: the variable is bound to a specific storage address, in which lies the value of the variable. We define the set of all possible storage addresses, which we will refer to as locations from now on, to be the set $\mathbb{N}$ of natural numbers. In order to map a variable name in the set *var-name* to a location we define the following set of partial functions:

$$Env_v = \mathit{var\text{-}name} \cup \{ \text{ next } \} \rightharpoonup \mathbb{N}$$

An element in Env_v is called env_v. The variable "next" is pointing to the next free location. This variable can be updated by using the function:

$$new(l) = l + 1, \text{ where } l \in \mathbb{N}$$

To update an environment, we use the notation $env_v[x \mapsto l]$, which is the environment env'_v defined by:

$$env'_v(y) = \begin{cases} env_v(y) & \text{if} \quad y \neq x \\ l & \text{if} \quad y = x \end{cases}$$

The notation $\mapsto$ is read "maps to". We define the set of storage locations, *Store*, to be the set of partial functions from locations to values:

$$Store = \mathbb{N} \rightharpoonup \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}$$

An element in *Store* is called *sto*. The notation for a storage update $sto[l \mapsto v]$, is the storage sto' defined by:

$$sto'(l') = \begin{cases} sto(l') & \text{if} \quad l' \neq l \\ l & \text{if} \quad l' = l \end{cases}$$

As was described in section 3.4.4 on page 27, an ID-list can be seen as a grouped declaration of variables, where each variable is identified by a key, $v \in \mathbb{R}$, instead of a name. We will need a partial function to store this information and we design a new environment, Env_i, which is defined by:

$$Env_i = \mathbb{N} \times \mathbb{R} \rightharpoonup \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}$$

An element in Env_i is called env_i. It can be updated using the notation $env'_i = env_i[n_1, v_1 \mapsto v]$ defined by:

$$env'_i(n_2, v_2) = \begin{cases} env_v(n_2, v_2) & \text{if} \quad n_2, v_2 \neq n_1, v_1 \\ v & \text{if} \quad n_2, v_2 = n_1, v_1 \end{cases}$$

The reason for having the set of $\mathbb{N}$ included in the input domain of the function is to have a way of uniquely identifying each ID-list from another. Why we have chosen the set of natural numbers instead of, for instance, the name of the variables which store the lists, will become apparent when we return to ID-lists in section 4.6.11 on page 70 where we give a formal definition of its behaviour.

We will now define an environment for method and event declarations:

$$Env_m = \textit{var-name} \rightharpoonup C \times \textit{var-name}^* \times Env_v \times Env_i$$

env_m is an element in Env_m. This means a method or event declaration is made from a command C (which may be composed of several other commands), a set of parameter names and a set of variables and ID-lists which were known at the time the method was declared. It can be updated using the notation $env'_m = env_m[x \mapsto \{C, \overrightarrow{y}, env_v, env_i\}]$ defined by:

$$env'_m(y) = \begin{cases} env_m(y) & \text{if } y \neq x \\ \{C, \overrightarrow{y}, env_v, env_i\} & \text{if } y = x \end{cases}$$

Methods will be explained in section 4.6.7 on page 62. Note that this way of modelling method and event declerations mean that, using this semantical definition, the programmer will be able to make a call to an event handler. We do not allow this behaviour in the language, but for simplicity of the rules we will not concider this. If we wanted to make sure this is impossible, we would need to define a new environment env_e where event declerations should be stored, exactly like with env_m.

4.2.2 The Stack

In small-step semantics there is no history of states that could be used to remove variables declared in a scope when leaving the scope. We solve this by creating a stack on which we put the environments. We will name this stack *envl* ('l' because it is a list of elements), which is an element in the set of stacks *Envl*, where:

$$Envl = (Env_v \times Env_m \times Env_i)^*$$

This stack is a set of variable, method and ID-list environments. Note that method environments will be the same in all elements of the stack since LagNa only allow methods and events to be declared in the same outermost scope. We specify that the stack *envl* has the set (env_v, env_m, env_i) as its top element by:

$$envl = (env_v, env_m, env_i) : envl'$$

We will need a construct to specify that a specific piece of code belongs to a certain set of environments from the stack. We model this by introducing a new construct **localscope** C **end** to our commands:

$$C \in commands \ \cup \ \text{localscope } C \text{ end}$$

4.3 Transition Systems

A transition system is a 3-tuple $(\Gamma, \rightarrow, T)$, where Γ is a set of configurations, $\rightarrow$ the transition rules and $T \subseteq \Gamma$ a set of end configurations. These configurations, or program states, are defined by a set of the environments and stores as described previously. Transition rules will be defined by a set of transition rules of the form:

$$[\text{Rule name}] \quad \frac{premises}{conclusion}, \quad \text{if} \quad conditions$$

Each transition rule is written as a number of *premisses* above a horizontal line, with a single *conclusion* below the line. When all of the premises are satisfied, the conclusion must hold. If a transition rule has no premises, it is called an *axiom* and we will leave out the horizontal line. Each rule has a name which we write on the left side. When needed, conditions restricting the applicability of the rule, as well as abreviations used within the rule, are annotated next to it.

Each rule describes a single step in the execution of a command. Each step may result in a change in the store or the environments (e.g. if it contains a method call). Therefore, we will use the following notation to describe a single step:

$$\langle C, sto, envl\rangle \Rightarrow \langle C', sto', envl'\rangle$$

4.4 Control Flow

LagNa is an imperative programming language (see section 2.2.1), however, it may be interrupted by events. The order in which commands are evaluated in a program may thus vary from each execution, given the environment which the game engine supplies. It is nessecary to understand how this flow in the code works, in order to understand the semantic rules. Figure 4.1

shows an example of how a program could be executed. We have marked certain points in the execution with numbers, which we will use to explain what happens at these key points.

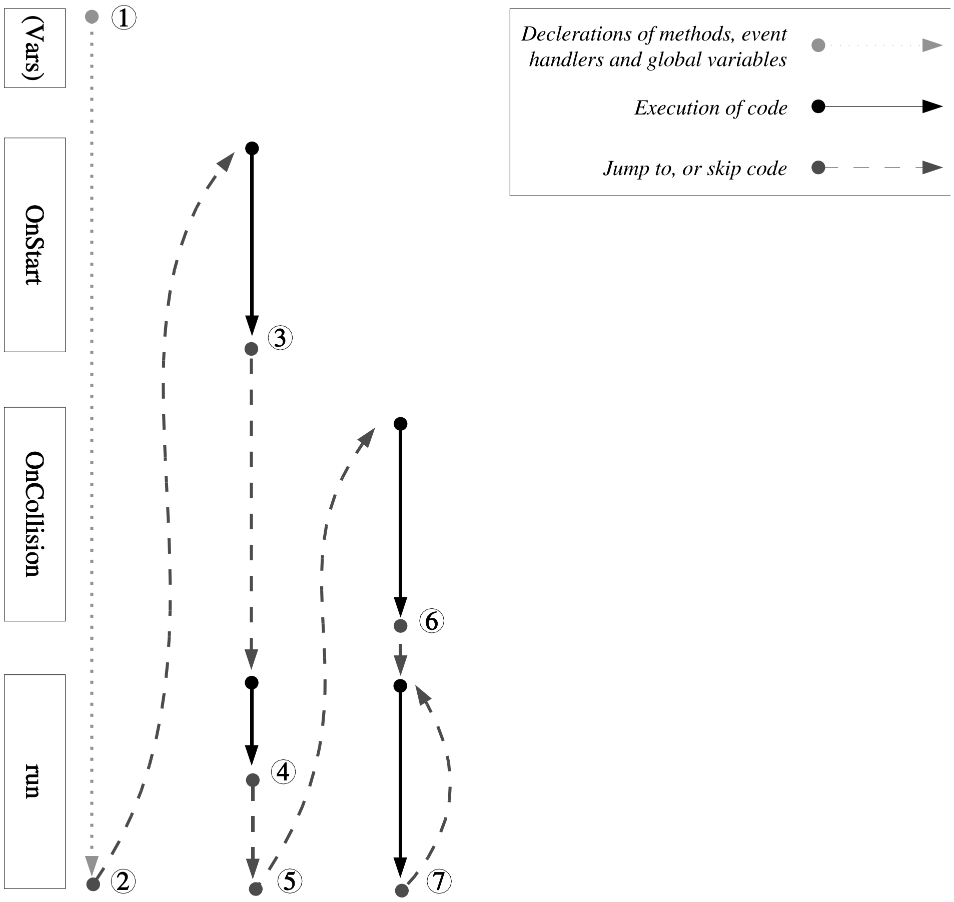


Figure 4.1: An example of how a program may behave under certain conditions

In this example the user has defined some global variables and three event handlers.

① When the program starts, the first thing it does is to define all global variables, method signatures and event signatures for later use (a signature is a mapping from identifiers to types). The program cannot be interrupted in this step, even if, theoretically, it should take a long time to complete, which means events could occur during this step. In reality this will never be an issue due to fast execution speed.

② Once all signatures have been established the program will enter a certain event-handling mechanism. This purpose of this mechanism is to check if there is an event waiting to be executed, and if not, execute the default handle called *run*. At this point in the program, the event-handler will find an *OnStart* event, and it will then execute the body of the event of the same name. From now on the program may be interrupted at any place in the code.

③ Once the execution of the body of *OnStart* has completed (in this case it was not interrupted), it returns to the event-checker. It will find that there are no events waiting, and so it executes the body of the default handler *run*.

④ During the execution of *run*, an *OnCollision* event occurs. All of the remaining commands will be skiped, and the program returns to the event-checker.

⑤ The event-checker finds the *OnCollision* event, and executes the body of the event.

⑥ Once this is done (there was no interruption, nor will there be any more) the program again returns to the event-checker

⑦ Assuming no more events occurs in the program, the event-checker will keep calling the *run* method over and over again. This is not an unlikely senario if the user does not specify any behaviour in the *run* method, since two of such bots in a battle could be standing still, waiting for each other to generate an event. This stresses the nessecity of specifying default behaviour in a bot, or at least never to let it stand still waiting for events.

When evaluating an expression, it will eventually evaluate to a value. If an event occurs during this evaluation, we cannot let it perform a simple skip, since that would put the program in an unspecified state. Therefore we will introduce a special value called EV which has no purpose but to ensure we can still use our semantic rules when an event has interrupted an expression. For instance, using the addition rule, $v_1 + v_2$ can still be used, whereas we would have to define an extra rule for $v+$ if an expression would simply skip when interrupted. The mechanism is a bit more complex than this, for instance $2 + EV$ still does not make any sense, but we will create conditions to capture this case.

To sum up, the control flow is sequencial, letting the control flow into sub-constructs from top to buttom, as long as each terminates normally. If an event occures while evaluating a cunstruct, the program will skip both remaining sub-constructs and enclosing constructs until a special construct handles the event, thereby resuming normal control flow from a specific point in the code. The program will never terminate by itself since it will always loop back to the *run* method if no event has occured.

4.5 Information Flow

The communication between the bot and the game engine is modeled through variables which can be set to certain values in the memory. For instance, when the bot need to check if an event has occured, it will dereference the variable *event*. When the bot needs to give information to the engine, such as when placing an action in a queue, we will put the value of the action in a specific place in the store (which is an abstraction of the memory) and update a variable which points to this new value. It is assumed that the game engine monitors these pointer variable all the time to make sure it will react instantly to any modifications.

How all this is actually done is more or less irrelevant in this context. Neither will we take synchronisation into acount, since this way of modeling the comunication is done in a different abstraction level then what is actually happening. Once the program has been compiled to Java Byte Code, which is required for a bot to work in GamNa, communication will often be nothing more than a call to the appropriate application interface of either the bot or the engine. These technical details are not of interrest at this moment, since we only want to describe the *behaviour* through an abstraction of how it is actually implemented.

4.6 Transition Rules

In the following section we will describe how the different constructs from our syntax, when executed, modify the state of the machine. We start by giving semantics rules for the basic types in our language. It is important to emphasise the distinction between *tokens* from our syntax and the *numerals* on which we can perform arithmetic operations in our abstract machine. Therefore we define the following literal functions:

We use the double brackets $[\![ ]\!]$to describe that the function takes syntactic arguments as input, and not literals. For instance the token "3.14" corresponds to the numeral $3.14 \in \mathbb{R}$ and we write: $\mathcal{R}\,[\![3.14\,]\!]= 3.14$. For the boolean truth values, we define the rules shown in Table 4.4

$$\mathcal{R} : real \rightarrow \mathbb{R}$$
$$\mathcal{B} : bool \rightarrow \mathbb{B}$$
$$\mathcal{S} : string \rightarrow \mathbb{S}$$

Table 4.3: Literal functions

$$\mathcal{B}\ [\![\text{true}]\!] = tt$$
$$\mathcal{B}\ [\![\text{false}]\!] = ff$$

Table 4.4: Boolean literals

Last the string function $\mathcal{S}$ will return the corresponding sequence of characters which makes up the string, so $\mathcal{S}\ [\![\text{“bot”}]\!] = \{\text{‘b’, ‘o’, ‘t’}\}$.

We will also use the set of litterals defined in Table 4.5 in the communication with the engine. Since they are not used specifically in the language, there is no need to create functions for mapping syntax, like with the other litterals.

Actions	Events
stop, left, right, forward, backward	OnStart, OnSight
backOrForth, leftOrRight	OnCollision, OnHit
chooseweapon, shoot	OnTargetHit, none

Table 4.5: Communication literals

The “Event” litterals are the ones the engine uses to specify whether an event has occured or not. The litterals in the “Movement” category will be described as we need them in the semantic rules.

The semantic rules shown in table 4.6 are used to evaluate literals into values, this done according to the types.

4.6.1 Arithmetic Expressions

It is possible to use simple mathematical operations $(+, -, *, /)$ with the type “real” in our language. The semantics rules, which describe how these mathematical operations behave, are shown in Table 4.7 on page 49.

As shown on these semantics clauses we use e for arithmetic expressions and v for values. As shown on Table 4.2 on page 40, e may be an expression of any kind. If e is *not* an arithmetic expression (it is either a boolean

[Booleans]	$bool \Rightarrow v$ if $\mathcal{B}[\![bool]\!] = v$
[Reals]	$real \Rightarrow v$ if $\mathcal{R}[\![real]\!] = v$
[Strings]	$string \Rightarrow v$ if $\mathcal{S}\ [\![string]\!] = v$

Table 4.6: Semantic rules for literal functions

expression or a string expression) there will not be any usable rules, and the configuration is stuck.

The fact that we define the rules using small-step semantics means we must evaluate the two expressions on each side of the operator one step at a time. It is always the leftmost expression which is evaluated first, and only when it is evaluated fully will the rightmost expression be evaluated. In an arithmetic expression it is not possible to update the variable, method or ID-list environments, which is why *envl* is the same both before and after the evaluation. The store, however, may be updated if the expression contains a method-call.

The rule names with the postfix "E" denotes that the rules descibes what happens in case an event occurs. We will use this notation throughout the this section for transision rules. In this case, if an event occurs, we will return the special "Event Value". As stated earlier, this is because another construct is expecting a return value of some kind. If an expression did not return anything, the configuration gets stuck.

Table 4.8 shows the semantic rules for evaluating brackets. With brackets it is possible to change the normal operator precedence (for normal operator precedence see Section 3.4.5 on page 28). Example 4.1 shows how the operator precedence can be changed.

Example 4.1.

$$\begin{aligned} 2 \cdot (2+5) &= 2 \cdot 7 = 14 \qquad \textit{With brackets} \\ &\ \textit{or} \\ 2 \cdot 2 + 5 &= 4 + 5 = 9 \qquad \textit{Without brackets} \end{aligned}$$

◆

[Addition-1] $$\frac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{\langle e_1 + e_2, sto, envl\rangle \Rightarrow \langle e_1' + e_2, sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Addition-2] $$\frac{\langle e_1, sto, envl\rangle \Rightarrow v}{\langle e_1 + e_2, sto, envl\rangle \Rightarrow \langle v + e_2, sto, envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Addition-1-2E] $\langle e_1 + e_2, sto, envl\rangle \Rightarrow EV$
if $env_v \circ sto(event) \neq \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Addition-3] $$\frac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle v + e, sto, envl\rangle \Rightarrow \langle v + e', sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Addition-4] $$\frac{\langle e, sto, envl\rangle \Rightarrow v}{\langle v + e, sto, envl\rangle \Rightarrow \langle v + v, sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Addition-3-4E] $\langle v + e, sto, envl\rangle \Rightarrow EV$
if $env_v \circ sto(event) \neq \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Addition-5] $\langle v_1 + v_2, sto, envl\rangle \Rightarrow v$
if $v_1 \neq EV \vee v_2 \neq EV$
where $v = v_1 + v_2$

[Addition-5E] $\langle v_1 + v_2, sto, envl\rangle \Rightarrow EV$
if $v_1 = EV \vee v_2 = EV$

Table 4.7: Addition operation. The semantic rules for substraction, multiplication and division are similar.

4.6.2 String Expressions

The only operation allowed in a string expression is the & symbol, which concatenates two values of any type (except ID-lists), and produces a string

[Brackets-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle (e), sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Brackets-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v}{\langle (e), sto, envl\rangle \Rightarrow v}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Brackets-1-2E]	$\langle (e), sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.8: Transition rules for brackets.

of the concatenation. The semantic rules for this operation is shown in Table 4.9. The values takes the form of expressions which needs to be evaluated before we can work on them. The first two rule continues to evaluate the first expression, until it returns a value. This is assuming no events has occured. In that case the [Concatenation-1-2E] rule will return the special "event value". Lastly the [Concatenation-5] rule will eventually return the actual concatenation of the two values of the expressions. We use the notation v_1v_2 for this concatenation of two sequences (consisting of characters).

4.6.3 Boolean Expressions

As written in the syntax (Chapter 3) it is possible in our language to use boolean expressions, even though these expressions would be meaningless for the language if they stood alone. like on Listing 4.4

```
run()
{
   2 < 3;
}
```

Listing 4.4: Boolean Expressions

This small code example does nothing which could benefit the user, but combined with other commands they would make a difference. We have the following boolean operations:

[Concatenation-1]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{\langle e_1 \ \&\ e_2, sto, envl\rangle \Rightarrow \langle e_1' \ \&\ e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Concatenation-2]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow v}{\langle e_1 \ \&\ e_2, sto, envl\rangle \Rightarrow \langle v \ \&\ e_2, sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Concatenation-1-2E]	$\langle e_1 \ \&\ e_2, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Concatenation-3]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle v \ \&\ e, sto, envl\rangle \Rightarrow \langle v \ \&\ e', sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Concatenation-4]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_2}{\langle v_1 \ \&\ e, sto, envl\rangle \Rightarrow \langle v_1 \ \&\ v_2, sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Concatenation-3-4E]	$\langle v \ \&\ e, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Concatenation-5]	$v_1 \ \&\ v_2 \Rightarrow v$ if $v_1 \neq EV \vee v_2 \neq EV$ where $v = v_1 v_2$
[Concatenation-5E]	$v_1 \ \&\ v_2 \Rightarrow EV$ if $v_1 = EV \vee v_2 = EV$

Table 4.9: More arithmetic operations

- Equal (=) this operation will compare two expressions and return true if both expressions gives the same value.

- Different from (<>) works completely opposite of Equal (=), so it would return true if the two expressions does not evaluate to the same value.
- Smaller than (<) will return true if the expression on the left side returns a smaller value than the right side expression.
- Larger than (>) does the opposite of smaller than (<)
- Smaller than, or equal to (<=) is combination between smaller than (<) and equal (=).
- Larger than, or equal to (>=) is combination between larger than (>) and equal (=).
- Negation (NOT) this operator will take an input and return true, if the input is false, and reverse when the input is true.
- And-expression (AND) validate two expressions and if both is true, AND will return true, else it will return false.
- Or-expression (OR) also validate two expressions, but it will return true if just one of the expressions is true, else false.

In the following we will describe all of the semantic rules with these operations with regards to their functionality.

Equality operations

As shown in the semantics on Table 4.10 and Table 4.11, [Equal-1], [Equal-2], and [Equal-3]-[Equal-4] is used to evaluate the two expressions until both have been evaluate to values. [Equal-5] and [Equal-6] determins if these two values of the expressions are equal. If they are, the whole construct will evaluate to tt (true), else it will evaluate to ff (false).

Besides = it is also possible to determine if a value is different from (<>) greater than (>), smaller than (<), smaller than or equal to (<=) or greater than or equal to (>=) another value. However the semantics for these operators is almost identical to equal (=), and therefore these have been omitted. All of these operations are also used in simple mathematics, although (<>) looks different than in normal mathematics ($\neq$). This, however, is only because ($\neq$) does not exist on a normal keyboard, but even though (<>) looks different than ($\neq$) they still have exactly the same behaviour. The operators >, <, >= and <= only works on reals, where <> and = works both on reals and booleans.

[Equal-1]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{\langle e_1 = e_2, sto, envl\rangle \Rightarrow \langle e_1' = e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Equal-2]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow v}{\langle e_1 = e_2, sto, envl\rangle \Rightarrow \langle v = e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Equal-1-2E]	$\langle e_1 = e_2, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.10: Boolean expressions for the *equal* operator, part 1.

NOT operation

The next semantic clauses in Table 4.12 only works with boolean type input, just as > and < only worked with reals. The operation is the *NOT* operation, normally typed as $\neg$ if used in mathematical expressions. This operation, given an arithmetical expression as input, returns anything else than the input. However because only one return value can be returned, this would be imposible to use, unless it should work as a random function, because what should NOT(42) return? Anything else than 42 would be right. Because we belive this behaviour of the NOT operator to be illogical, it will only work on boolean expression in LagNa. The reason this works is that boolean only have two terminals (true and false) so anything else than true have to be false and the opposite with false as input value.

The semantics shown on Table 4.12 takes a boolean expression as input (e) and evaluates it to a value. If the value is *tt* (true), [Not-3] will be used and the output will be *ff* (false). If the result is *ff* (false), [Not-4] will be used.

AND and OR operations

With the semantics already declared it is possible to compare two expressions and get a return value, but if the return value should be generated from more than two expression some other operation has to be used. This could be the *if* operation but this would generate large code. It will therefore not be easy to read or write the code with use of the if operation, as described in our

[Equal-3]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle v = e, sto, envl\rangle \Rightarrow \langle v = e', sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Equal-4]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_2}{\langle v_1 = e, sto, envl\rangle \Rightarrow \langle v_1 = v_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Equal-3-4E]	$\langle v = e, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Equal-5]	$\langle v_1 = v_2, sto, envl\rangle \Rightarrow tt$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v_1 = v_2$
[Equal-6]	$\langle v_1 = v_2, sto, envl\rangle \Rightarrow ff$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v_1 \neq v_2$
[Equal-5-6-E]	$\langle v_1 = v_2, sto, envl\rangle \Rightarrow EV$ if $v_1 = EV \vee v_2 = EV$

Table 4.11: Boolean expressions for the *equal* operator, part 2. This is similar to the "<>", ">", ">=", "<", and "<=" operations.

quality goals in Section 2.1 on page 7, that it should. A small example of how the code could possibly look with extensive use of the if operator is shown on Listing 4.5.

Because of this we have chosen to implement two operations to avoid this large amount of code, the two operations is AND and OR with these two operations the example would be different, as we show on Listing 4.6.

The semantic rules in Table 4.13 show that if both expressions return true (if $v_1 = tt \wedge v_2 = tt$) the output will also be true. If one of the expressions returns false the output will be false; it will also return false if both of the expressions are false. Other than that, they are similar to the NOT semantics.

The OR operation works almost the same way. The only difference is that the OR operation only need one of the two expressions to evaluate to

[Not-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{NOT } e, sto, envl\rangle \Rightarrow \langle \text{NOT } e', sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Not-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{NOT } e, sto, envl\rangle \Rightarrow \langle \text{NOT } v, sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Not-1-2E]	$\langle \text{NOT } e, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Not-3]	$\langle \text{NOT } v, sto, envl\rangle \Rightarrow \mathit{ff}$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v = \mathit{tt}$
[Not-4]	$\langle \text{NOT } v, sto, envl\rangle \Rightarrow \mathit{tt}$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v = \mathit{ff}$
[Not-3-4E]	$\langle \text{NOT } v, sto, envl\rangle \Rightarrow EV$ if $v = EV$

Table 4.12: Boolean expressions for negation (NOT) operator

true. If both expression return true, OR will also return true.

These are all the boolean operations which can be used in our language. As described earlier some of the operations are only in the language to ease up the writing and reading of the code. These operations have been implemented to fulfill our quality goals (see Section 2.1 on page 7).

4.6.4 Variables

In this section we will describe the decleration of variables, assignment to variables and lastly dereferencing a variable.

As the syntax describe in section 3.4 on page 25, the characteristics for an assignment is ":=" which could look like the example on Listing 4.7:

We will start with assignment, which is shown in Table 4.15. When assigning an expression to a variable, the expression must first be evaluated,

```
real number := 2;

run()
{
  if 1 < number
  {
    if 3 > number
    {
      if number <> 2.5
      {
        if number = 2
        {
          do stuff;
        }
        if number = 2.01
        {
          do stuff;
        }
      }
    }
  }
}
```

Listing 4.5: Example of code without use of AND and OR

```
real number := 2;

run()
{
  if (1 < number) AND (3 > number) AND (number <> 2.5) AND ((number
      = 2) OR (number = 2.01))
  {
    do stuff;
  }
}
```

Listing 4.6: Example of code with use of AND and OR

```
real number;
run()
{
  number := 40+2;
```

Listing 4.7: Example of syntactically correct assignment

which is what happens in rule [Assignment-1] and [Assignment-2]. This is assuming no events has occured. If so, we will not return EV as we have done previously. The reason is that an assignment is a command, and thus it will not return a value, but instead the changes store and possibly a changed $envl$. Therefore, if an event occurs, the [Assignment-1-2E] rule will just return the same store and $envl$, which means nothing will be changed. Also, it is no longer enough to check if any event has occured. This is because the OnStart

[And-1]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{e_1 \text{ AND } e_2 \Rightarrow \langle e_1' \text{ AND } e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[And-2]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow v}{\langle e_1 \text{ AND } e_2\rangle \Rightarrow \langle v \text{ AND } e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[And-1-2E]	$\langle e_1 \text{ AND } e_2, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[And-3]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle v \text{ AND } e\rangle \Rightarrow \langle v \text{ AND } e', sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[And-4]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_2}{\langle v_1 \text{ AND } e\rangle \Rightarrow \langle v_1 \text{ AND } v_2, sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[And-3-4E]	$\langle v \text{ AND } e, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[And-5]	$\langle v_1 \text{ AND } v_2, sto, envl\rangle \Rightarrow tt$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v_1 = tt \wedge v_2 = tt$
[And-6]	$\langle v_1 \text{ AND } v_2, sto, envl\rangle \Rightarrow ff$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v_1 = ff \wedge v_2 = ff$
[And-5-6-E]	$\langle v_1 \text{ AND } v_2, sto, envl\rangle \Rightarrow EV$ if $v_1 = EV \vee v_2 = EV$

Table 4.13: Boolean expressions for the AND operator.

[Or-1]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{e_1 \text{ OR } e_2 \Rightarrow \langle e_1' \text{ OR } e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Or-2]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow v}{\langle e_1 \text{ OR } e_2\rangle \Rightarrow \langle v \text{ OR } e_2, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Or-1-2E]	$\langle e_1 \text{ OR } e_2, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Or-3]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle v \text{ OR } e\rangle \Rightarrow \langle v \text{ OR } e', sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Or-4]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_2}{\langle v_1 \text{ OR } e\rangle \Rightarrow \langle v_1 \text{ OR } v_2, sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Or-3-4E]	$\langle v \text{ OR } e, sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Or-5]	$\langle v_1 \text{ OR } v_2, sto, envl\rangle \Rightarrow tt$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge (v_1 = tt \vee v_2 = tt)$
[Or-6]	$\langle v_1 \text{ OR } v_2, sto, envl\rangle \Rightarrow ff$ if $v_1 \neq EV \wedge v_2 \neq EV \wedge v_1 = ff \wedge v_2 = ff$
[Or-5-6-E]	$\langle v_1 \text{ OR } v_2, sto, envl\rangle \Rightarrow EV$ if $v_1 = EV \vee v_2 = EV$

Table 4.14: Boolean expressions for the OR operator.

event will always be thrown in the very beginning by the engine (and only once), but we need to declare all global variables (i.e. they must not be skiped).

Once the expression has been evaluated to a value, assuming no events has occured, the [Assignment-3] will return the same *envl* and an updated store, where the location l of the variable x is mapped to the value e.

$$\text{[Assignment-1]} \quad \frac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle x := e, sto, envl\rangle \Rightarrow \langle x := e', sto', envl\rangle}$$
$$\begin{aligned}&\text{if } env_v \circ sto(event) = \text{none}\\ &\vee\ env_v \circ sto(event) = \text{OnStart}\\ &\text{where } envl = (env_v, env_m, env_i) : envl'\end{aligned}$$

$$\text{[Assignment-2]} \quad \frac{\langle e, sto, envl\rangle \Rightarrow v}{\langle x := e, sto, envl\rangle \Rightarrow \langle x := v, sto, envl\rangle}$$
$$\begin{aligned}&\text{if } env_v \circ sto(event) = \text{none}\\ &\vee\ env_v \circ sto(event) = \text{OnStart}\\ &\text{where } envl = (env_v, env_m, env_i) : envl'\end{aligned}$$

$$\text{[Assignment-1-2E]} \quad \begin{aligned}&\langle x := e, sto, envl\rangle \Rightarrow (sto, envl)\\ &\text{if } env_v \circ sto(event) \neq \text{none}\\ &\wedge\ env_v \circ sto(event) \neq \text{OnStart}\\ &\text{where } envl = (env_v, env_m, env_i) : envl'\end{aligned}$$

$$\text{[Assignment-3]} \quad \begin{aligned}&\langle x := v, sto, envl\rangle \Rightarrow (sto[l \mapsto v], envl)\\ &\text{if } env_v \circ sto(event) = \text{none}\\ &\vee\ env_v \circ sto(event) = \text{OnStart}\\ &\text{where } envl = (env_v, emv_m, env_i) : envl'\\ &\text{and } l = env_v(x)\end{aligned}$$

$$\text{[Assignment-3E]} \quad \begin{aligned}&\langle x := v, sto, envl\rangle \Rightarrow (sto, envl)\\ &\text{if } env_v \circ sto(event) \neq \text{none}\\ &\wedge\ env_v \circ sto(event) \neq \text{OnStart}\\ &\text{where } envl = (env_v, env_m, env_i) : envl'\end{aligned}$$

Table 4.15: Transition rules for the assignment command.

We will now describe the decleration of a variable, which is shown in Table 4.16. We have used the notation *id* which is the same as defined in the CFG. It can be either of the terminals "real", "boolean" or "string". To declare a variable x without initialisation, we will need to update the variable environment env_v. We do this by mapping x to the next free location l which is found by $l = env_v(next)$. Now that we have used the "next free location", we also need to update this. Therefore, the *next* variable is mapped to the following allocation in the store, using the *new* function on l. This is what happens in [Variable-dec-1].

To declare a variable with initialisation, we will use make use of three other rules: first we will declare the variable without initialisation, then we will use the composition rule (which will be described later), and finally we will use the assignment rule to actually assign the expression to the variable. The reason we split the construct up like this is that we would otherwise need a lot of rules to describe each step in the evaluation, and we have already created rules that can be used. This is shown in the [Variable-dec-2] rule.

[Variable-dec-1]	$\langle id\ x, sto, envl\rangle \Rightarrow (sto,$ $(env_v[x \mapsto l][next \mapsto new(l)], env_m, env_i) : envl')$ if $env_v \circ sto(event) =$ none $\vee\ env_v \circ sto(event) =$ OnStart where $envl = (env_v, env_m, env_i) : envl'$ and $l = env_v(next)$
[Variable-dec-1E]	$\langle id\ x, sto, envl\rangle \Rightarrow \langle sto, envl\rangle$ if $env_v \circ sto(event) \neq$ none $\wedge\ env_v \circ sto(event) \neq$ OnStart where $envl = (env_v, env_m, env_i) : envl'$
[Variable-dec-2]	$\langle id\ x := e, sto, envl\rangle \Rightarrow \langle id\ x; x := e;, sto, envl\rangle$ if $env_v \circ sto(event) =$ none $\vee\ env_v \circ sto(event) =$ OnStart where $envl = (env_v, env_m, env_i) : envl'$
[Variable-dec-2E]	$\langle id\ x := e, sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq$ none $\wedge\ env_v \circ sto(event) =$ OnStart where $envl = (env_v, env_m, env_i) : envl'$

Table 4.16: Transition rules for the declaring a variable.

Lastly we will describe how to dereference a variable, which is shown in

Table 4.17. It only consist of one small rule, which is nothing more than to use the function $env_v \circ sto(x)$ which will return the value bound to x.

[Variable-dereference]	$\langle x, sto, envl\rangle \Rightarrow v$ if $env_v \circ sto(x) = v$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.17: Transition rules for dereferencing a variable.

4.6.5 Composition Rules

The next semantic rule will describe how two commands after each other will behave. This is necessary to describe, because command number two should continue with the changes the first command made. The rules are shown in Table 4.18.

[Composition-1]	$\dfrac{\langle C_1, sto, envl\rangle \Rightarrow \langle C_1', sto', envl'\rangle}{\langle C_1\ C_2, sto, envl\rangle \Rightarrow \langle C_1'\ C_2, sto', envl'\rangle}$
[Composition-2]	$\dfrac{\langle C_1, sto, envl\rangle \Rightarrow (sto', envl')}{\langle C_1\ C_2, sto, envl\rangle \Rightarrow \langle C_2, sto', envl'\rangle}$
[Composition-3]	$\langle C;, sto, envl\rangle \Rightarrow \langle C, sto, envl\rangle$

Table 4.18: Transition rules for the composition command.

To achieve this behavior C_1 is evaluated first, which is what happens in [Composition-1]. The store and the *evnl*, which might have been changed, is then used when evaluation C_2. The [Composition-3] rule simply ignores the ';' in the end of a command, if it is present (for instance, a while command does not end with a semicolon).

4.6.6 Active Code

As we described in Section 4.2.2 on page 42, we need the special construct "localscope C end" to specify what element in the stack C belongs to. Each time we enter a new scope, we will thus encapsulate the commands in that scope in this constuct. This is done when evaluating an "if", "case" or "while" structure[1] and a method call.

Once a command has been encapsulated by the "localscope ... end", two things can happen. If C is evaluated to a new C' (it was not completely evaluated in this one step), it will just continue evaluating C' with the possibly changes store and environments. This is what happens in [Composition-1]. If the evaluation of C is completed, which means the tuppel $(sto', envl')$ is returned, it means we are done with all commands in this scope. We then return the possibly changed store along with the environments just below the one currently used. This happens in [Composition-2]. In both these cases it is provided that no events will occur. Should an event occur, the command will simply be skiped, and the top environments is popped from the stack. This happens in [Composition-3].

[Scope-1]	$\dfrac{\langle C, sto, envl\rangle \Rightarrow \langle C', sto', envl''\rangle}{\langle \text{localscope } C \text{ end}, sto, envl\rangle \Rightarrow \langle \text{localscope } C' \text{ end}, sto', envl''\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl''$
[Scope-2]	$\dfrac{\langle C, sto, envl\rangle \Rightarrow (sto', envl'')}{\langle \text{localscope } C \text{ end}, sto, envl \Rightarrow (sto', envl')}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Scope-3]	$\langle \text{localscope } C \text{ end}, sto, envl\rangle \Rightarrow (sto, envl')$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.19: Transition rules for commands in a scope.

4.6.7 Methods

In LagNa it is possible to declare methods and call already declared methods. This gives the possibility of reusing code. As described in Section 3.6 on page 30, any number of parameters may be specified in a method.

LagNais defined using static binding of methods, which in principle is impossible, because methods do not need to be declared before use in LagNa,

[1]note that the semantics for "while" does not explicitly create a "localscope". Instead the structure is rewritten as an "if" structure, which then create the "localscope".

which is a requirement for making static binding. However, as we will see later, we can obtain this behaviour by letting the semantics make a pass of the program, where is collects all method (and event) declarations, so they are known before the actual execution of the program. This also have the effect that we do not need to update the method environment when calling recursive methods, making recursive methods a part of LagNa.

We are using the call-by-value parameter mekanism for our methods. This basically means that each parameter passed to the function is copied to a new local variable in the body of the function, and that the function only changes the copies and not the original parameters.

[Method-dec]	$\langle identifier\ M(identifier\ x_1, \ldots, identifier\ x_n)\ \{C\}, sto, envl\rangle \Rightarrow$ $(sto, (env_v, env_m[M \mapsto (C, \overrightarrow{x}, env_v)], env_i) : envl')$ where $\overrightarrow{x} = (x_1, \ldots, x_n)$ and $envl = (env_v, env_m, env_i) : envl'$
[Event-dec]	$\langle event\ M(identifier\ x_1, \ldots, identifier\ x_n)\ \{C\}, sto, envl\rangle \Rightarrow$ $(sto, (env_v, env_m[M \mapsto (C, \overrightarrow{x}, env_v)], env_i) : envl'$ where $\overrightarrow{x} = (x_1, \ldots, x_n)$ and $envl = (env_v, env_m, env_i) : envl'$
[Method-call-1]	$\langle M(\overrightarrow{e}), sto, envl\rangle \Rightarrow \langle \text{localscope } x_1 := e_1; \ldots; x_n := e_n;\ C \text{ end},$ $sto, (env'_v[\overrightarrow{x} \mapsto \overrightarrow{l}][next \mapsto new(l_n)], env_m, env_i) : envl\rangle$ if $env_v \circ sto(event) = \text{none}$ where $l = env_v(next)$ and $env_m(M) = (C, \overrightarrow{x}, env'_v)$ and $l_1 = l, l_2 = new(l_1), \ldots, l_n = new(l_{n-1})$ and $envl = (env_v, env_m, env_i) : envl'$
[Method-call-1E]	$\langle M(\overrightarrow{e}), sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.20: Transition rules for method calls

4.6.8 The If Statement

As shown in earlier code examples, see Listing 4.5 on page 56 and 4.6 on page 56, it is possible to use conditional commands in LagNa. With these conditional commands it is possible to execute a command if a boolean expression is true or another one if the boolean expression is false. The rules which describe how an *if* statement works are shown in Table 4.21. It will evaluate a certain command C if and only if a boolean expression e is true. The rules for the *if-else* statement are shown in Table 4.22. If the expression e evaluates to tt (true), it will evaluate the first command C_1, if it evaluates to ff (false), the second command C_2 will be evaluated.

[If-1]
$$\frac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{if } e\ \{C\}, sto, envl\rangle \Rightarrow \langle \text{if } e'\ \{C\}, sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $(env_v, env_m, env_i) : envl'$

[If-2]
$$\frac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{if } e\ \{C\}, sto, envl\rangle \Rightarrow \langle \text{if } v\ \{C\}, sto, envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $(env_v, env_m, env_i) : envl'$

[If-1-2E] $\langle \text{if } e\ \{C\}, sto, envl\rangle \Rightarrow (sto, envl)$
if $env_v \circ sto(event) \neq \text{none}$
where $(env_v, env_m, env_i) : envl'$

[If-3] $\langle \text{if } v\ \{C\}, sto, envl\rangle \Rightarrow$
$\langle \text{localscope } C \text{ end}, sto, (env_v, env_m, env_i) : envl\rangle$
if $v = tt$

[If-3E] $\langle \text{if } v\ \{C\}, sto, envl\rangle \Rightarrow (sto, envl)$
if $v = ff \vee v = EV$

Table 4.21: Transition rules for the conditional command *if*.

First the boolean expression is evaluated ([If-1]), then it checks if it is true of false [If-2] or [If-2E]. If the boolean expression is true then the command C is evaluated. If the boolean expression is false, or an event has happened, then *if* does nothing (it leads to the same sto and envl).

The [If-else] rule works much like [If]. The difference is that [If-else] handle the ‘else’ construct of an if construct, where the [If] semantic does not. The ‘else’ construct needs to be evaluated whenever the boolean expression is false [If-else-3], and then C_2 is evaluated instead of C_1.

[If-else-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{if } e\ \{C_1\} \text{ else } \{C_2\}, sto, envl\rangle \Rightarrow \langle \text{if } e'\ \{C_1\} \text{ else } \{C_2\}, sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $(env_v, env_m, env_i) : envl'$
[If-else-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{if } e\ \{C_1\} \text{ else } \{C_2\}, sto, envl\rangle \Rightarrow \langle \text{if } v\ \{C_1\} \text{ else } \{C_2\}, sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $(env_v, env_m, env_i) : envl'$
[If-else-1-2E]	$\langle \text{if } e\ \{C_1\} \text{ else } \{C_2\}, sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $(env_v, env_m, env_i) : envl'$
[If-else-3]	$\langle \text{if } v\ \{C_1\} \text{ else } \{C_2\}, sto, envl\rangle \Rightarrow$ $\langle \text{localscope } C_1 \text{ end}, sto, (env_v, env_m, env_i) : envl\rangle$ if $v = tt$
[If-else-4]	$\langle \text{if } v\ \{C_1\} \text{ else } \{C_2\}, sto, envl\rangle \Rightarrow$ $\langle \text{localscope } C_2 \text{ end}, sto, (env_v, env_m, env_i) : envl\rangle$ if $v = ff$
[If-else-3-4E]	$\langle \text{if } v\ \{C\} \text{ else } \{C_2\}, sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$

Table 4.22: Transition rules for conditional command *if-else*

4.6.9 The While Statement

As seen in the syntax LagNa also has a "while" command. This command could be used to repeat some parts of the code a number of times. This means that it is possible to run a single or more commands without having to write the commands more than once. This is illustrated in Listing 4.8 and 4.9

The code in Listing 4.9 will do the same as the code in Listing 4.8 but with the use of while loops it is much easier and faster to write the code. Also the code written will be more clear, because it will be more compact. But because there are no rules for how many times a while loop may repeat itself, it will be possible to make while loops which will take very long time

```
real number := 4;
run()
{
  number := number - (4/3);
  number := number + (4/5);
  number := number - (4/7);
  number := number + (4/9);
  .
  .
  .
  number := number - (4/1337);
}
```

Listing 4.8: Calculation without while loops

!htb

```
real number := 4;
real i := 3;
run()
{
  while(i<1337)
  {
    number := number - (4/i);
    i := i + 2;
    if i < 1337
    {
      number := number + (4/i);
      i := i + 2;
    }
}
```

Listing 4.9: Calculation with while loops

to complete or maybe never will complete ("while(true) { }"). But because GamNa is a event based game, it is possible to break out of while loops before the boolean expression is evaluated to false. The while loop is divided into *if* constructs, and as long as the *if* construct is evaluated to true, the [While] semantics will be evaluated again (see Table 4.21 for the rules for the if construct). If the [If] rule evaluates the boolean expression to false, or an event occurs, then the while construct stops running. The semantics for the *while* command is shown in Table 4.23.

[While]	$\langle \text{while } e \ \{C\}, sto, envl\rangle \Rightarrow$ $\langle \text{if } e \ \{C \text{ while } e \ \{C\}\}, sto, envl\rangle$

Table 4.23: Transition rules for the while statement

4.6.10 The Select Statement

The select statement in LagNa , as described in Section 3.7.2 on page 32, is slightly complex to describe formally since there are many ways of expressing a "case". We will now go through each of these possibilities using the syntactic categories found in Chapter 3 on page 19.

In a select statement there is given an arithmetic expression, e, which may or may not need to be evaluated to a value. This must only be done once in the evaluation of the whole select statement since an arithmetic expression may produce side effects. This is shown in Table 4.24.

[Select-1]
$$\frac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{select } e \text{ \{ } cases \text{ \}}, sto, envl\rangle \Rightarrow \langle \text{select } e' \text{ \{ } cases \text{ \}}, sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Select-2]
$$\frac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{select } e \text{ \{ } cases \text{ \}}, sto, envl\rangle \Rightarrow \langle \text{select } v \text{ \{ } cases \text{ \}}, sto, envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Select-1-2E] $\langle \text{select } e \text{ \{ } cases \text{ \}}, sto, envl\rangle \Rightarrow (sto, envl)$
if $env_v \circ sto(event) \neq \text{none}$

Table 4.24: Transition rules for the select construct.

The *cases* category consists of a number of "cases". Each case may also consist of several intervals, the value v must be within this range. We will first look at the circumstance where a *case* is made up of first an interval joined with more intervals. This is shown in Table 4.25 on the next page. Here we need to test whether or not v lies in the first interval. If it does, we need to evaluate the rest of the intervals as shown in the first rule [Select-1]. If it does not, we should skip this *case* and continue with the next as shown in the last rule [Select-2].

If the first interval is actually a number, we only need to test whether or not it is the same number as v. Apart from this, Table 4.26 is similar to the table before.

If there was, or is now, only one interval left, we need to test whether the value v is in this interval. If it is, the command belonging to the case is evaluated and the select statement is done. If v is not in the interval, we will skip to the next case. This is shown in table 4.27 on page 69

[Select-3]	$\langle \text{select } v \ \{ \ [v_1 : v_2], \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle \Rightarrow$ $\langle \text{select } v \ \{ \ \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle$ if $v \neq EV$ and $v >= v_1 \wedge v <= v_2$
	We will omit the 3 similar rules for $]v_1 : v_2],]v_1 : v_2[$ and $[v_1 : v_2[$. These determine whether the comparisons should be sharp or not.
[Select-4]	$\langle \text{select } v \ \{ \ [v_1 : v_2], \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle \Rightarrow$ $\langle \text{select } v \ \{ \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle$ if $v \neq EV$ and $v < v_1 \vee v > v_2)$
	Again we will omit the 3 similar rules.
[Select-3-4-E]	$\langle \text{select } v \ \{ \ [v_1 : v_2], \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle \Rightarrow$ $(\textit{sto}, \textit{envl})$ if $v = EV$

Table 4.25: Transition rules for the select construct with a start interval, with an undefined number of case expressions following.

[Select-5]	$\langle \text{select } v \ \{ \ v_1, \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle \Rightarrow$ $\langle \text{select } v \ \{ \ \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle$ if $v \neq EV$ and $v = v_1$
[Select-6]	$\langle \text{select } v \ \{ \ v_1, \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle \Rightarrow$ $\langle \text{select } v \ \{ \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle$ if $v \neq EV$ and $v \neq v_1$
[Select-5-6-E]	$\langle \text{select } v \ \{ \ v_1, \textit{case-exp} \ \{C\} \ \textit{cases} \ \}, \textit{sto}, \textit{envl} \rangle \Rightarrow$ $(\textit{sto}, \textit{envl})$ if $v = EV$

Table 4.26: Transition rules for the select construct with a start number, with an undefined number of case expressions following.

If the interval was actually a number, we will just check if v is the same

[Select-7]	$\langle \text{select } v \; \{ \; [v_1 : v_2] \; \{C\} \; cases \; \}, sto, envl\rangle \Rightarrow$ $\langle \text{localscope } C \text{ end}, sto, (env_v, env_m, env_i) : envl\rangle$ if $v \neq EV$ and $v >= v_1 \wedge v <= v_2$
	Again we will omit the 3 similar rules.
[Select-8]	$\langle \text{select } v \; \{ \; [v_1 : v_2] \; \{C\} \; cases \; \}, sto, envl\rangle \Rightarrow$ $\langle \text{select } v \; \{ \; cases \; \}, sto, envl\rangle$ if $v \neq EV$ and $v < v_1 \vee v > v_2)$
	Again we will omit the 3 similar rules.
[Select-7-8-E]	$\langle \text{select } v \; \{ \; [v_1 : v_2] \; \{C\} \; cases \; \}, sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$

Table 4.27: Transition rules for the select construct, where the select case ends with an interval.

number. Other than that, Table 4.28 is similar to the one before.

[Select-9]	$\langle \text{select } v \; \{ \; v_1 \; \{C\} \; cases \; \}, sto, envl\rangle \Rightarrow$ $\langle \text{localscope } C \text{ end}, sto, (env_v, env_m, env_i) : envl\rangle$ if $v \neq EV$ and $v = v_1$
[Select-10]	$\langle \text{select } v \; \{ \; v_1, \{C\} \; cases \; \}, sto, envl\rangle \Rightarrow$ $\langle \text{select } v \; \{ \; cases \; \}, sto, envl\rangle$ if $v \neq EV$ and $v \neq v_1$
[Select-9-10-E]	$\langle \text{select } v \; \{ \; v_1 \; \{C\} \; cases \; \}, sto, envl\rangle \Rightarrow$ $(sto, envl)$ if $v = EV$

Table 4.28: Transition rules for the select construct, where the select case ends with a number

We continue to evaluate each case in the select statement until we either find one that matches or there are non left. If no case was found, but a *default* case exists, we will evaluate the belonging command. If there is no default case, the select statement will exit. This is shown in Table 4.29. Note that the evaluation of e might still have produced side-effects which has altered the store.

[Select-10]	$\langle \text{select } v \ \{ \text{ default } \{C\} \ \}, sto, envl\rangle \Rightarrow$ $\langle \text{localscope } C \text{ end}, sto, (env_v, env_m, env_i) : envl\rangle$
[Select-10E]	$\langle \text{select } v \ \{ \text{ default } \{C\} \ \}, sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$
[Select-11]	$\langle \text{select } v \ \{\}, sto, envl\rangle \Rightarrow (sto, envl)$
[Select-11E]	$\langle \text{select } v \ \{\}, sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$

Table 4.29: Transition rules for the the select construct, with the default case

4.6.11 ID-Lists

As we described in Section 3.4.4 on page 27, an ID-list is a container of values, where each value is mapped to a key in the set of integer numbers. We store this information in the evironment env_i. We also need a way of uniquely identifying each ID-list from each other. The most logical way of doing this might be to define the environment as:

$$Env_i = \textit{var-name} \times \mathbb{R} \rightharpoonup \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}$$

That is, given a variable name (the name of the ID-list) and a Real number (the key), the function will return the value bound to this set of information. However, a variable name does not uniquely identify an ID-list in the env_i environment. This becomes evident when overriding a name of an ID-list in a different scope. The problem is that this gives access to previously defined keys, which belongs to a different ID-list in a different scope. This means that if the programmer accidentally tries to derefference a key which does not exist, but the key was initialized in an ID-list with the same name in a previous scope, the program will not return an error but a value which may or may not be of the same type. Worse, the programmer might try and change this value, but in fact a different ID-list will be changed. This means an error like this will be difficult to locate. It is now clear that a different form of identifying the ID-lists from each other is needed. If we look at how a non-list variable is declared, we see that it is mapped to a new and empty memory location. We can use this location as identifier, and so the environment becomes:

$$Env_i = \mathbb{N} \times \mathbb{R} \rightharpoonup \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}$$

That is, given a memory location and a Real number, the value bound to this set is returned. We store this number of the location in env_v just like when declaring and initializing a variable like we saw in the [Variable-dec]rule in Section 4.6.4 on page 55. The only difference is that instead of storing a value, we store the location of itself. This can be seen in Table 4.30. Note that *id* is the syntactic categori defined in CFG 3.3.

[List-dec]	$\langle id[]\ x, sto, envl\rangle \Rightarrow$ $(sto[l \mapsto l], (env_v[x \mapsto l][next \mapsto new(l)], env_m, env_i) : envl')$ if $env_v \circ sto(event) = \text{none} \vee env_v \circ sto(event) = \text{OnStart}$ where $envl = (env_v, env_m, env_i) : envl'$ and $l = env_v(next)$
[List-dec-E]	$\langle id[]\ x, sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none} \wedge env_v \circ sto(event) \neq \text{OnStart}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.30: Transition rules for ID-lists

When assigning a key in the ID-list to a value, both the key and the value may be an arithmetic expression which needs to be evaulated to litterals. This is what happens in Table 4.31 in [List-key-assignment-1] to [List-key-assignment-4]. Once this is done, we update env_i to this new mapping of a location and a key to a value, which we explained before.

To dereference a key in an ID-list, we first need to evaluate the key, which is shown in Table 4.32. If no event occured, we return the value which is bound the the tupel $(location, key)$ as shown in [List-deref-3].

It is also possible to assign a whole ID-list to another ID-list. This is done the exact same way as for variables of different types, as we defined in Section 4.6.4 on page 55 and because of that, there is no need to create additional rules.

Note that this way of constructing the ID-lists will make assignments of ID-lists behave differently than that of other types. This is because the

[List-key-assignment-1]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{\langle x[e_1] := e_2, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}$ if $env_v \circ sto(event) =$ none where $envl = (env_v, env_m, env_i) : envl'$
[List-key-assignment-2]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow v}{\langle x[e_1] := e_2, sto, envl\rangle \Rightarrow \langle x[v] := e_2, sto, envl\rangle}$ if $env_v \circ sto(event) =$ none where $envl = (env_v, env_m, env_i) : envl'$
[List-key-assignment-1-2E]	$\langle x[e_1] := e_2, sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq$ none where $envl = (env_v, env_m, env_i) : envl'$
[List-key-assignment-3]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle x[v] := e, sto, envl\rangle \Rightarrow \langle x[v] := e', sto', envl\rangle}$ if $env_v \circ sto(event) =$ none where $envl = (env_v, env_m, env_i) : envl'$
[List-key-assignment-4]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_2}{\langle x[v_1] := e, sto, envl\rangle \Rightarrow \langle x[v_1] := v_2, sto, envl\rangle}$ if $env_v \circ sto(event) =$ none where $envl = (env_v, env_m, env_i) : envl'$
[List-key-assignment-3-4E]	$\langle x[v] := e, sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq$ none where $envl = (env_v, env_m, env_i) : envl'$
[List-key-assignment-5]	$\langle x[v_1] := v_2, sto, envl\rangle \Rightarrow$ $(sto, (env_v, env_m, env_i[(l, v_1) \mapsto v_2]) : envl')$ if $env_v \circ sto(event) =$ none where $envl = (env_v, env_m, env_i) : envl'$ and $l = env \circ sto(x)$
[List-key-assignment-5E]	$\langle x[v_1] := v_2, sto, envl\rangle \Rightarrow (sto, envl)$ if $v_1 = EV \vee v_2 = EV$

Table 4.31: Transition rules for ID-lists

[List-deref-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle x[e], sto, envl\rangle \Rightarrow \langle x[e'], sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[List-deref-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v}{\langle x[e], sto, envl\rangle \Rightarrow \langle x[v], sto, envl\rangle}$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[List-deref-1-2E]	$\langle x[e], sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[List-deref-3]	$\langle x[v_1], sto, envl\rangle \Rightarrow v_2$ if $v_1 \neq EV$ where $envl = (env_v, env_m, env_i) : envl'$ and $v_2 = env_i((l, v))$ and $l = env_v \circ sto(x)$
[List-deref-3E]	$\langle x[v], sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$

Table 4.32: Transition rules for ID-lists

ID-lists are implemented as call-by-refence in the style Java uses.

Its is not possible to make a clone of a list, by assigning it to another empty list.

If we wanted call-by-value for the ID-lists done the same way as the other types in the language, we would need to modify the semantics for variable assignment to check which type of value it is assigned to. If the value is a number in a set of natural numbers (i.e. it is a reference), it must make a copy of all the mappings in the ID-list which the reference points to.

4.6.12 Continual Actions

Besides the common commands, LagNa also have a set of actions commands which indicate how a bot should run and act. An action could example be rotate(r), where r is a Real, which would tell the bot that it should rotate r

degrees in the direction specified by the sign of r.

We have two ways of calling actions. One where we indicate how long the action stretches over, and another where the action runs until explicitly stopped by the programmer. These two ways of calling an action work somewhat differently in the semantics. By making a call to a continual action, all that needs to be done is to set a variable which is monitored by the engine. Specified actions, on the other hand, have queue systems. These are slightly more complex and we will treat them in the next section (Section 4.6.13 on page 76). First we will descibe the continual actions.

When a continual action is called, an appropriate variable is set to the value determined by what the action was. The engine will keep performing the action until the variable is set to the value *stop*. It is also possible to suspend the action and resume it later. This happens automatically if there are anything in the relevant queue. For instance, the programmer may call the continual move-action *moveforward* in the beginning of the program. From now on the bot will always move forward, until told otherwise. Later the programmer calls the specified move-action called *move(-50)*, which means the bot will now ignore the continual move-action, and perform a backwards movement of 50 units. Once this is accomplished, it will resume the continual moveforward() action. Another way of understanding continual actions is to see it as a *default* behaviour when nothing else is specified.

The first continual actions we will descibe are the movement actions, which can be seen in Table 4.33. These actions tell the bot in which of the four directions it should move when nothing is specified in the movement queue. The first rule evaluates a "moveforward()" construct, simply by setting the "move-repeat" variable to "forward", asuming no events has occured. In that case the second rule will just skip the action entirely. These two rules are similar to "movebackwards()", "strafeleft()" and "straferight()".

This means it is not possible to move either back or forth while strafing at the same time, since all these actions uses the same variable and queue system. If two of these actions are called one directly after the other, the last action will override the first one. An example of this is shown in Listing 4.10.

```
run()
{
  moveforward();
  strafeleft();
}
```

Listing 4.10: Example of usage of continual actions in LagNa

The piece of code shown on Listing 4.10 will first set the "move-repeat" variable to "forward", but right after the "strafeleft()" action will change this variable to "left" instead. The reason these actions are not placed in the queue with the specified actions are because it run indefinitely, never allowing the next action in the queue to be executed.

[Moveforward-1]	$\langle \text{moveforward}(), sto, envl \rangle \Rightarrow$ $(sto[\text{move-repeat} \mapsto \text{forward}], envl)$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Moveforward-1E]	$\langle \text{moveforward}(), sto, envl \rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.33: Transition rules for continual move actions. The rules for move-backwards, strafeleft and straferight are similar, except that move-repeat will be mapped to the litterals: "backward", "left" and "right", respectively

The next action is continual rotate which could for example be used if a programmer wants the bot to walk around in circles. The semantics are similar to the movement actions, apart from the fact that it is no longer the "move-repeat" variable which is set, but one called "rotate-repeat". The rules can be found in Table 4.34.

[Rotateleft-1]	$\langle \text{rotateleft}(), sto, envl \rangle \Rightarrow$ $(sto[\text{rotate-repeat} \mapsto \text{left}], envl)$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Rotateleft-1E]	$\langle \text{rotateleft}(), sto, envl \rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.34: Transition rules for continual rotate actions. Rotateright is similar, except rotate-repeat will map to "right" instead

The semantical rules for look, and shoot is identical, but it uses the look-repeat, and shoot-repeat variables.

4.6.13 Specified Actions

We will now introduce the queue system for specified actions. A queue works by the first-in-first-out principle. Calling a specified action will put the argument to the action in the relevant queue, and when the action is in the front, the engine will start executing it. Once the action has been performed fully, it will be removed from the queue and the game engine will start executing the action placed just after. If no actions are in the queue, it will see if a continual action has been specified.

An example to illustrate this can be seen in Listing 4.11.

```
run()
{
  strafeleft();
  move(5);
  move(-5);
}
```

Listing 4.11: Example of usage of queues in GamNa

This small piece of code will first tell the bot that unless otherwise specified, it should always strafe left. Next in the code, it tells the engine to move the bot forward 5 units by putting "move(5)" in the move queue. After this the bot is told to move backwards 5 units, and "move(-5)" is placed in the queue. The program will continue with whatever lines of code is following, which is all done in an instant compared to how long it takes the engine to perform all the calculations for all the objects in the game. This is just to make it clear that the mechanism that points to what line of code which need to be executed does not necessarily correspond to what action the engine is currently performing for the bot. A specified action does not "pause" the program counter until the action is completed since this would make make simultanious actions impossiple. This is the reason for the queue system.

Returning to the example, when the engine is performing the specified actions for the bot, it will first check if there are any actions in the move-queue. There it finds the "move(5)" action in front in the queue. This tells the engine that the bot should move forward 5 units. When this is done, "move(5)" will be removed from the queue and now the "move(-5)" action is in front in the queue. This tells the engine that the bot should move backwards 5 units, and after this, the "move(-5)" action is removed from the queue. Now there are no more items in the queue, and so the engine checks if there are any continous action specified. There it finds that the bot should strafe left until the continous action is changed, stopped, or new actions are placed in the queue.

This performance could also have been done without a queue if the program counter waited for move(5) to finish before reading any further. But because we wanted to have the opportunity to execute other commands while the bot moves, such as rotating a specific number of degrees, or shooting a few times while moveing. This will be impossible without queues for all the different types of actions. This means we will define the following queues:

- Move
- Rotate
- Look
- Shoot

Actions in different queues will be performed simultaneously. This can be shown by making a little change to the example from before

```
run()
{
  move(5);
  move(-5);
  rotate(10);
}
```

Listing 4.12: Example of usage of more then one queues in GamNa

We remove the "strafeleft()" action, and ss before, "move(5)" and "move(-5)" will be put on the queue. But directly after, the "rotate(10)" action will be put in to the rotate queue. This means that "move(5)" and "rotate(10)" will now be executed simultaniously. There is no guerantee that moving one unit takes the same amount of time in the engine as rotating one degree, but to make things simple, we will asume this is the case. This means that when "move(5)" is completed, "rotate(10)" still needs to rotate for 5 degrees more. The "move(-5)" action will then be executed meanwhile the "rotate(10)" action is still being executed.

This might seem confusing at first, but we believe that the programmer will quickly be comfortable with this through experimentation with the different actions.

Before we introduce the semantic rules, we need to clearify how we specify the simultaneous behaviour. Throughout the semantical rules, we have tried to only to descibe how the language itself behaves, leaving out what actually might happen in the game engine. For instance, in the language, we do not keep track of where the bots are possisioned in the battlefied, in order to throw events when they collide with something. This would be to big a task, and it would not yield any useful information since this is all handled by the engine which means it does not have be taken into acount when constructing the compiler. Likewise, all simultaneous actions are handled by the engine, and we will model it here. As state earlier in this section, the program counter moves down in a sequential order, never pausing even when performing an action. All which happens is the specified action is placed in the queue, and the program moves to the next command.

Now we are ready to introduce the way we have construced the queues. Recalling that the store is a sequence of memory allocations, we will reserve an amount for each queue, each one starting with two variables denoting the head and the tail of the queue. The tail always points to the next empy location in the queue. When the head variable points to the same location as the tail variable, it means there are no actions in the queue. When putting an action in the queue, we place it in the location which the tail points to, and make the tail point to the location just after using the "new" function. It is up to the game engine to move the head pointer when an action is completet. Later we will introduce the "stop" action which alters this explicitly though.

We will wait with the description of how this reservation is actually done. For now we just asume that nothing will interfere with the store in this sequence of allocations. To distinguish the different queues from eachother, we will give the "head" and the "tail" variables a postfix denoting what queue they belong to. The postfixes we will use are: "mq" for movement queue, "rq" for rotate queue, "lq" for the look queue and "sq" for the shoot queue. For instance, the tail of the movement queue will be called "tail-mq".

The first specified action we will describe is for rotating a specific number of degrees. The semantic rules can be seen in Table 4.35. The first two rules are for evaluating the argument in the form of an expression. When this is done the [Rotate-degrees-3] rule will place the value of the expression in the location l which is the next free location in the queue, found from the location that tail-rq is pointing to. Then tail-rq is updated to the next location. This is assuming no event has happened. If the expression returned the "event value", it means the command must be skipped, which is what happens in the last rule.

We will now descibe the movement actions "move(real)" and "strafe(real)". They are slightly different from the before mentioned actions. In this case, it is not enough to place the value of the argument in the queue, since it will not be clear whether it means the bot should move back or forth, or left or right. To specify this, we will reserve two slots in the queue for one action. The first slot can have the values "backOrForth" or "leftOrRight", and the second slot holds the value. Other than this, the rules in Table 4.36 is similar

[Rotate-degrees-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{rotate}(e), sto, envl\rangle \Rightarrow \langle \text{rotate}(e'), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Rotate-degrees-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{rotate}(e), sto, envl\rangle \Rightarrow \langle \text{rotate}(v), sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Rotate-degrees-1-2E]	$\langle \text{rotate}(e), sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Rotate-degrees-3]	$\langle \text{rotate}(v), sto, envl\rangle \Rightarrow (sto[l \mapsto v],$ $(env_v[\text{tail-rq} \mapsto new(l)], env_m, env_i) : envl')$ if $v \neq EV$ where $envl = (env_v, env_m, env_i) : envl'$ and $l = env_v \circ sto(\text{tail-qr})$
[Rotate-degrees-3E]	$\langle \text{rotate}(v), sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$

Table 4.35: Transition rules for specified rotating. The semantic rules for looking are similar except the queue ”tail-ql” is used instead.

to the ones we described before for rotating.

The last semantic rules we will descibe for specified actions is for shooting and for choosing weapons. These actions are placed on the same queue since it will not make sence to change weapon without knowing how many “bullets” are left in the shoot queue. We model this in the exact same way as we did for movement: by making two slots for each action. This time the first slot will indicate if the action is a weapon selection (the value will then be set to “chooseweapon”) or a number of times to fire the selected weapon (the value will then be set to “shoot”). The semantic rules for choosing a weapon can be seen in Table 4.37 on page 81. It is similar in structure to the rules for specified movement, which we covered earlier. The weapons which are available to choose from is denoted by a number. A reference table to what number is mapped to what weapon can be found in Table A on page 125.

[Move-distance-1]
$$\frac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{move}(e), sto, envl\rangle \Rightarrow \langle \text{move}(e'), sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Move-distance-2]
$$\frac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{move}(e), sto, envl\rangle \Rightarrow \langle \text{move}(v), sto, envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Move-distance-1-2E]
$\langle \text{move}(e), sto, envl\rangle \Rightarrow (sto, envl)$
if $env_v \circ sto(event) \neq \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Move-distance-3]
$\langle \text{move}(v), sto, envl\rangle \Rightarrow$
$(sto[l \mapsto \text{`}backOrForth'][l' \mapsto v],$
$(env_v[\text{tail-mq} \mapsto new(l')], env_m, env_i) : envl')$
if $v \neq EV$
where $envl = (env_v, env_m, env_i) : envl'$
and $l = env_v \circ sto(\text{tail-mq})$ and $l' = new(l)$

[Move-distance-3E]
$\langle \text{move}(v), sto, envl\rangle \Rightarrow (sto, envl)$
if $v = EV$

Table 4.36: Transition rules for specified forwards or backwards movement. Strafing is completely similar except rule three will store “leftOrRight” instead of “backOrForth”.

We now give the semantics for the mathmatical functions supported by LagNa. We start with the random function. To make it easier for ourselves we define a semantic helper function called *rand*:

$$rand : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

The *rand* function returns a random value that range within the specified arguments. eg. $rand(3, 6)$ can return a value in the interval $[3; 6]$. The semantic is shown at Table 4.38

We define the helper functions for sine:

$$sine : \mathbb{R} \to \mathbb{R} cosine : \mathbb{R} \to \mathbb{R}$$

[Weapon-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \text{chooseweapon}(e), sto, envl\rangle \Rightarrow \langle \text{chooseweapon}(e'), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Weapon-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v}{\langle \text{chooseweapon}(e), sto, envl\rangle \Rightarrow \langle \text{chooseweapon}(v), sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Weapon-1-2E]	$\langle \text{chooseweapon}(e), sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Weapon-3]	$\langle \text{chooseweapon}(v), sto, envl\rangle \Rightarrow (sto[l \mapsto \text{`}chooseweapon'\text{'}]$ $[l' \mapsto v], (env_v[\text{tail-sq} \mapsto new(l')], env_m, env_i) : envl')$ if $v \neq EV$ where $envl = (env_v, env_m, env_i) : envl'$ and $l = env_v \circ sto(\text{tail-sq})$ and $l' = new(l)$
[Weapon-3E]	$\langle \text{chooseweapon}(v), sto, envl\rangle \Rightarrow (sto, envl)$ if $v = EV$

Table 4.37: Transition rules for specified forwards or backwards movement. Strafing is completely similar except rule three will store “shoot” instead of “chooseweapon”

The *sine* and *cosine* functions behaves like the normal mathemathical sine and cosine functions. See Table 4.39 for the semantic for Sine.

In LagNa it is also possible to use the mathemathical functions “square root” and “power”. See the semantic at Table 4.40 and Table 4.41.

In LagNa three special actions exist, these actions is special because they return a value. These three actions are: “gethealth()”,“getviewingangle()” and “canshoot()”. We define the helper function *health* which take nothing as input and return the health of the bot. These can be seen in Table 4.42.

[Random-1] $$\frac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{\langle \mathrm{random}(e_1, e_2), sto, envl\rangle \Rightarrow \langle \mathrm{random}(e_1', e_2), sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Random-2] $$\frac{\langle e_1, sto, envl\rangle \Rightarrow v_1}{\langle \mathrm{random}(e_1, e_2), sto, envl\rangle \Rightarrow \langle \mathrm{random}(v_1, e_2), sto, envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Random-1-2E] $\langle \mathrm{random}(e_1, e_2), sto, envl\rangle \Rightarrow EV$
if $env_v \circ sto(event) \neq \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Random-3] $$\frac{\langle e_2, sto, envl\rangle \Rightarrow \langle e_2', sto', envl\rangle}{\langle \mathrm{random}(v_1, e_2), sto, envl\rangle \Rightarrow \langle \mathrm{random}(v_1, e_2'), sto', envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Random-4] $$\frac{\langle e_2, sto, envl\rangle \Rightarrow v_2}{\langle \mathrm{random}(v_1, e_2), sto, envl\rangle \Rightarrow \langle \mathrm{random}(v_1, v_2), sto, envl\rangle}$$
if $env_v \circ sto(event) = \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Random-3-4E] $\langle \mathrm{random}(v_1, e_2), sto, envl\rangle \Rightarrow EV$
if $env_v \circ sto(event) \neq \text{none}$
where $envl = (env_v, env_m, env_i) : envl'$

[Random-5] $\langle \mathrm{random}(v_1, v_2), sto, envl\rangle \Rightarrow v$
if $v_1 \neq EV \wedge v_2 \neq EV$
where $v = \mathrm{rand}(v_1, v_2)$

[Random-5E] $\langle \mathrm{random}(v_1, v_2), sto, envl\rangle \Rightarrow EV$
if $v_1 = EV \vee v_2 = EV$

Table 4.38: Transition rules for the *random* function.

4.6.14 Stopping and Waiting for Actions

The stop actions discontinues earlier called actions from running any further. It also empties the relevant queue. To stop an ongoing action, the stop action

[Sine-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \sin(e), sto, envl\rangle \Rightarrow \langle \sin(e'), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Sine-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_1}{\langle \sin(e), sto, envl\rangle \Rightarrow \langle \sin(v_1), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Sine-1-2E]	$\langle \sin(e), sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Sine-3]	$\langle \sin(v_1), sto, envl\rangle \Rightarrow v$ if $v \neq EV$ where $v = \text{sine}(v)$
[Sine-3E]	$\langle \sin(v_1), sto, envl\rangle \Rightarrow EV$ if $v = EV$ where $v = \text{sine}(v)$

Table 4.39: Transition rules for the *sine* function. The transition rules are similar for the *cosine* function.

sets the relevant action variable to “stop”. A queue is emptied by setting the tail of the queue to point to the head of the queue. The semantic rules for stopping an action can be seen in Table 4.43.

To stop all actions, the action simply called “stop” makes implicit calls to every other “stop” action, effectively making the bot do nothing at all. The semantic rules can be seen in Table 4.44.

The *wait* commands gives the programmer the ability to insure that an action queue has emptied before continuing evaluating the code. This means that wait commands are blocked commands. The semantic rules for waiting for movement can be seen in Table 4.45. The first rule keeps calling itself, assuming the head of the queue does not points to the tail of the queue, or an event has occured. We assume that this looping behaviour takes a certain amount of time. Then, at some point, the movement queue should be emptied, and/or an event occurs, effectively breaking the loop. This is

[Sqrt-1]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow \langle e', sto', envl\rangle}{\langle \mathrm{sqrt}(e), sto, envl\rangle \Rightarrow \langle \mathrm{sqrt}(e'), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Sqrt-2]	$\dfrac{\langle e, sto, envl\rangle \Rightarrow v_1}{\langle \mathrm{sqrt}(e), sto, envl\rangle \Rightarrow \langle \mathrm{sqrt}(v_1), sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Sqrt-1-2E]	$\langle \mathrm{sqr}(e), sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Sqrt-3]	$\langle \mathrm{sqrt}(v_1), sto, envl\rangle \Rightarrow v$ if $v \neq EV$ where $v = \sqrt{v}$
[Sqrt-3E]	$\langle \mathrm{sqrt}(v_1), sto, envl\rangle \Rightarrow EV$ if $v = EV$

Table 4.40: Transition rules for the *square root* function.

seen in the second rule.

The semantics for all the other wait commands are the same, except which queues they check if empty.

We also make it possible to either wait for an event, or to wait for all queues to be emptied. The rules for this is seen in Table 4.46. The only difference in these rules when checking for events, compared to waiting for movement as we described before, is a different side condition. The rule for waiting for all actions is simply an implicit call to all the wait commands.

4.6.15 Initialization and Event Handling

The purpose of the *check-event* structure is to check if an event has happened, and if so, run the relevant event code. The semantic rules is shown in Table 4.47 on page 88. If no events has happened, the "run" method is called, which is what happens in the first rule. Otherwise, the relevant event handler is called, and the event indicator is set to "none" to prevent the body

[Power-1]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow \langle e_1', sto', envl\rangle}{\langle \text{power}(e_1, e_2), sto, envl\rangle \Rightarrow \langle \text{power}(e_1', e_2), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Power-2]	$\dfrac{\langle e_1, sto, envl\rangle \Rightarrow v_1}{\langle \text{power}(e_1, e_2), sto, envl\rangle \Rightarrow \langle \text{power}(v_1, e_2), sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Power-1-2E]	$\langle \text{power}(e_1, e_2), sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Power-3]	$\dfrac{\langle e_2, sto, envl\rangle \Rightarrow \langle e_2', sto', envl\rangle}{\langle \text{power}(v_1, e_2), sto, envl\rangle \Rightarrow \langle \text{power}(v_1, e_2'), sto', envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Power-4]	$\dfrac{\langle e_2, sto, envl\rangle \Rightarrow v_2}{\langle \text{power}(v_1, e_2), sto, envl\rangle \Rightarrow \langle \text{power}(v_1, v_2), sto, envl\rangle}$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Power-3-4E]	$\langle \text{power}(v_1, e_2), sto, envl\rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Power-5]	$\langle \text{power}(v_1, v_2), sto, envl\rangle \Rightarrow v$ if $v_1 \neq EV \wedge v_2 \neq EV$ where $v = v_1^{v_2}$
[Power-5E]	$\langle \text{power}(v_1, v_2), sto, envl\rangle \Rightarrow EV$ if $v_1 = EV \vee v_2 = EV$

Table 4.41: Transition rules for the *power* function.

of the event to be skiped, and to keep the event from being evaluated more than once. This is the only place where the event handlers are called.

[Gethealth-1]	$\langle \text{gethealth}(), sto, envl \rangle \Rightarrow v$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$ and $v = health$
[Gethealth-1E]	$\langle \text{gethealth}(), sto, envl \rangle \Rightarrow EV$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.42: Transition rules for the gethealth function, the transition is similar for getviewingangle and canshoot.

[Stop-move-1]	$\langle \text{stopmove}(), sto, envl \rangle \Rightarrow (sto[l_1 \mapsto \text{'stop'}][l_2 \mapsto l_3], envl)$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$ and $l_1 = env_v(\text{move-repeat})$ and $l_2 = env_v(\text{head-mq})$ and $l_3 = env_v(\text{tail-mq})$
[Stop-move-1E]	$\langle \text{stopmove}(), sto, envl \rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.43: Transition rules for stopping actions. Note that "stopmove" also stops strafing. Transition rules for *stoprotate*, *stoplook* and *stopshoot* are the same, with the exception of which queue they use.

We will now describe the last rule in the semantics, which is resonsible for setting up all the variables. It is shown in Table 4.48. Note that we have not defined a formal way of specifying that this is a rule which must be called as the very first one, and only once. We could do this by defining a special token which all programs must start with, but for simplicity we have left this out.

In the semantics for starting the program, we first declare all necessary helping variables, namely "event", which we have used for checking if an event has occured or not, "next", which we use to point to the next free location, and the "head" and "tail" variables used in the queue systems. The "event" variable is set to the first memory allocation in the store. This means the next

[Stop-all-1]	$\langle \text{stop}(), env_v, env_m, sto\rangle \Rightarrow$ $\langle \text{stopmove}(); \text{stoprotate}(); \text{stopturn}(); \text{stopshoot}(), sto, envl\rangle$ if $env_v \circ sto(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[Stop-all-1E]	$\langle \text{stop}(), env_v, env_m, sto\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(event) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.44: Transition rules for stopping actions. Note that stopmove also stops strafing

[WaitForMove-1]	$\langle \text{waitformove}(), sto, envl\rangle \Rightarrow \langle \text{waitformove}(), sto, envl\rangle$ if $env_v \circ sto(\text{event}) = \text{none} \wedge$ $env_v(\text{tail-mq}) \neq env_v(\text{head-mq})$ where $envl = (env_v, env_m, env_i) : envl'$
[WaitForMove-1E]	$\langle \text{waitformove}(),\ sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(\text{event}) \neq \text{none} \vee$ $env_v(\text{tail-mq}) = env_v(\text{head-mq})$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.45: Transition rules for waiting. The rules for "waitforrotate", "waitforlook" and "waitforshoot" are similar, except the use the appropriate queues instead

free location is the second allocation, which is what “next” is set to. Then each set of the queue variables is placed within a hundred allocations apart. This means that the semantics only allow up to 99 variables to be decleared before things might start to go wrong. It also means that no more than a hundred actions can ever be placed in each queue before the guarantee that they will not start overlapping disappears. This is of cause not a limitation we want to impose on the language, but for simplicity this is what we have written in the rule. Of cause it is easy to set this value much higher to allow for more room in each sequence in the memory, but since the semantics is described in a higher abstraction level than what can actually be achieved, we will rely on the Java Virtual Machine for handling the memory.

[WaitForEvent-1]	$\langle \text{waitforevent}(), sto, envl\rangle \Rightarrow \langle \text{wait}(), sto, envl\rangle$ if $env_v \circ sto(\text{event}) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[WaitForEvent-1E]	$\langle \text{wait}(), sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(\text{event}) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[WaitForActions-1]	$\langle \text{waitforactions}(), sto, envl\rangle \Rightarrow \langle \text{waitformove}();$ $\text{waitforrotate}();\ \text{waitforlook}();$ $\text{waitforshoot}(),\ sto, envl\rangle$ if $env_v \circ sto(\text{event}) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$
[WaitForActions-1E]	$\langle \text{waitforactions}(), sto, envl\rangle \Rightarrow (sto, envl)$ if $env_v \circ sto(\text{event}) \neq \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$

Table 4.46: Transition rules for commands

[check-event-none]	$\langle \text{check-event}, sto, envl'\rangle \Rightarrow$ $\langle C; \text{check-event}, sto, envl\rangle$ if $sto \circ env_v(event) = \text{none}$ where $envl = (env_v, env_m, env_i) : envl'$ and $env_m(run) = (C, env_v, env_i)$
[check-event-OnStart]	$\langle \text{check-event}, sto, envl'\rangle \Rightarrow$ $\langle C; \text{check-event}, sto[event \mapsto none], envl\rangle$ if $sto \circ env_v(event) = \text{OnStart}$ where $envl = (env_v, env_m, env_i) : envl'$ and $env_m(OnStart) = (C, env_v, env_i)$

Table 4.47: Transition rules for event checking. The rules for OnCollision, OnSight, OnHit, OnTargetHit and OnDeath are similar to OnStart, except the side condition and method look-ups are done with the coresponding names.

Secondly we set the *value* of the "event" variable, which is placed in the first allocation, to "OnStart".

Thirdly we run the whole program through by evaluating C. Note that we do not allow commands other than declarations, to be placed outside methods or events. This means that *only* the global variable declerations, method and event declerations will be stored in the relevant environments.

Finally the "check-event" construct is evaluated as we descibed earlier, which starts evaluating first the "OnStart" event, if present, and then the "run" method, if present, or any other events which might occur.

[program-start]	$\langle C, sto, envl\rangle \Rightarrow \langle C;\text{check-event}, sto[1 \mapsto \text{`}OnStart'],$ $(env_v[\text{event} \mapsto 1][\text{next} \mapsto 2][\text{head-mq} \mapsto 100]$ $[\text{tail-mq} \mapsto 100][\text{head-rq} \mapsto 200][\text{tail-rq} \mapsto 200]$ $[\text{head-lq} \mapsto 300][\text{tail-lq} \mapsto 300][\text{head-sq} \mapsto 400]$ $[\text{tail-sq} \mapsto 400], env_m, env_i) : envl'\rangle$ where C is the body of the program, and $envl = (env_v, env_m, env_i) : envl'$

Table 4.48: Transition rules for the structure to initialize the program

Advantage and disadvantage of Small-step semantics for LagNa

In this section we will describe the advantages and disadvantages of small-step semantics.

In GamNa an event call should preferably interrupt the running code instantly and take care of the event which have been called. This is possible by describing LagNa with small-step semantics, because small-step describes the individual steps of a command, therefore interrupts can occur between any two of these steps.

Instead of handling the event calls right away, LagNa could deal with the event as soon as possible. This would mean that LagNa should finish all the running code and after all code has been executed, it should take care of the event. This however would cause a small delay on when the event would be handled. This behaviour could be made by use of atomic small-step.

Listing 4.13 shows the delay an atomic small-step semantics for LagNa would cause. If the semantics for LagNa is a normal small-step semantics it will be possible to interrupt where the ¤ is, and directly jump to the OnHit event, had it been atomic small-step all the code under the ¤ would first have to be executed before the OnHit event could be handled.

```
1  run()
2  {
3    real number=0;
4
5    move(10);¤
6    ChooseWeapon(1);
7    While(number < 10)
8    {
9      rotate(36);
10     WaitForRotate;
11     Shoot(1);
12     number := number + 1;
13   }
14 }
15
16 event OnHit(real AngleFromSight)
17 {
18   say("Grrr be careful, I know where you live!!")
19   look(AngleFromSight);
20   Shoot(10);
21 }
```

Listing 4.13: An illustration of the delay an atomic small-step semantics for LagNa would causes (The ¤ is where the code gets interrupted by the OnHit event)

This would mean a delay of "1*ChooseWeapon(1); + 10*While(number < 10) + 10*4 (the inside of the while loop) = 51 commands/action" however these 51 commands/actions can not directly be compared, because some of the commands are faster to execute than others.

For instance line 10 and 11 in Listing 4.13 will have to wait for the bot to rotate 36° degrees, which is probably slower than the code shown at line 13 which only increase the variable "number" with one.

This delay would mean that while the 51 commands/action was being run through, the OnHit event might have been called again, or maybe the bot would die before it could react on the OnHit event. Had the code been using a "while(true)" command instead of "while(number < 10)" the bot would never complete the while loop, and therefore would never have the chance to react on the event.

Instead of using atomic small-step, normal small-step semantics could be used, however this would lead to problems with stack overflow, unless these problems are explicitly solved in the semantics. Normally when using small-step semantics, stack overflow is not an issue, because when a command, described using small-step semantics, gets interrupted and halt, to run another command, the computer will eventually come back to execute the rest of the interrupted code. However in LagNa this is not the case, because when a command is interrupted by an event, the event should be executed and thereafter the code should return to "Run()" instead of returning to the interrupted code.

The reason the program does not return to the interrupted code is because when a bot has handled an event, the other bots could have moved or the bot itself could have changed state during the event handling, which means the condition of GamNa is not the same as before the event occurred.

By interrupting code and never returning again, will fill up the stack. If an event was to interrupt an assignment like "x := x + 10;" right after the first x. The computer would then have x loaded into the memory because it "thinks" the value of x would have to be updated, but instead of updating the value of x, a jump in the code take place to take care of an event. Because the assignment never gets to finish, x would fill up a part of the memory in the stack. If this was to repeat itself over and over, the stack would end up being filled with variables, all ready to get their values updated, but because that would never happen, none of these variables would be freed again, thus creating a stack overflow.

This problem can be solved if the semantics instead of interrupting the code instantly, skips all the commands after an event call until the event is reached. This way, all of the different scopes will be unloaded. However this solution, will lead to strange variables, if for instance an assignment, which changes a global variable "x" to the return value of a method, and this method is interrupted during execution, causing all commands to be skipped. This would result in "x" never being assigned to anything.

This could be solved by only allowing events to interrupt between two commands (using the composition semantics), then we could ensure that the first command would be fully completed before the interrupt. This again leads to the problem of non-instant interruption of the code, and all other commands than composition would have to be either atomic small step (atomic means, non breakable) or big step.

For instance, continuing the example from before, with a variable "x" being assigned to the return value of a method, which we call *ReturnANumber()*. When doing this assignment, an event occurs while inside the actual method body, then the command which has already been called, will then be executed, but the following command will not be executed. This is due to the semantics rules of "Composition", which describes that the event interrupting the execution should be handled instead. But since the assignment rule would have to be atomic, it would not be possible to jump to the actual event before the assignment of "x" is complete.

This small piece of code (Listing 4.14) illustrates the problem of using small step for Composition and atomic small step for all other commands. When the event occurs the program is already inside a command (the x :=

```
run()
{
  real x := ReturnANumber();
}

method real ReturnANumber()
{
  real R:=0;¤
  while(true)
  {
    R := R + random(-10,10);
  }
  return 0;
}
```

Listing 4.14: problem with atomic small step (the ¤ indicate where the code gets interrupted by an event)

ReturnANumber() assignment) which is atomic, and there by impossible to interrupt, and because of this the semantics for composition will not have the ability to jump directly to the event which interrupted.

Another way of solving the problem could be to take care of event interrupts in all of the commands, this could be done by writing two semantic rules for every command, one rule for no event interruption and one rule if an event occurs. By doing this an assignment for instance would only assign a new value to a variable if no event has occurred, and if an event has occurred the variable will just keep the value it had before the assignment.

Returning to the initial problem with the stack being filled another solution could be to implement a garbage collector, in the GamNa engine, see [Seb04] section 6.9.9.3 for a description of garbage collecting. This garbage collector should however be a little special, because normally garbage collectors only clean memory cells which is not mapped to by any other memory cells. But because the computer still "thinks" all the interrupted assignments will finish at some point, all the memory cells are still mapped to by other memory cells and therefore a normal garbage collector would "think" the memory cells are still in use, and omit cleaning these cells. To solve this problem different technique's can be used, some however would be more efficient than others. One way to solve the problem would be to clean up all previous assignments when a new assignment is loaded into the memory, however some of the functionalities in LagNa would be lost , because in LagNa it is possible to assign a variable to the value of a method call, and in the method it is possible to initialize and assign new temporary variables, this would mean that a temporary variable assignment would override the one already taking place, and the return value of the method would then be lost. Listing 4.15 shows an assignment, with this kind of garbage collecting.

```
1  run()
2  {
3    real Bar := Nine();
4    real Temporary := foo(Bar);
5  }
6
7  method real Nine()
8  {
9    return 9;
10 }
11
12 method real foo(real R)
13 {
14   real Destroy := R+1;
15   return R;
16 }
```

Listing 4.15: An illustration of how a “clean earlier assignment” garbage collector would destroy assignments

Memmory Block	1	2	3	4
Variable in memmory	Bar	Temporary	R	Destroy
After line 3:	Bar :=	Nil	Nil	Nil
After line 4:	Bar := 9	Temporary :=	Nil	Nil
After line 9:	Bar = 9	Temporary :=	Nil	Nil
After line 14:	Bar = 9	Nil	R=9	Destroy := 9+1

Figure 4.2: Illustrate the values of the variables in 4.15

Figure 4.2 shows how the variables in Listing 4.15 is stored in the memory. Notice that the variable “Temporary” never gets a value assigned, even though the code in 4.15 describes that it should have a value, but because the garbage collector would remove all open assignments when a new assignment gets executed, the assignment with the variable “Temporary”, stored in the memory, will be removed by the garbage collector, and the return value of the method “foo()” will never be used.

Instead of removing all open assignments the garbage collector could suffice with cleaning up only the “old” open assignments. But to locate these “old” open assignments it would be necessary to keep some kind of time stamp on all memory blocks. This method would probably be more efficient than cleaning all open assignments, because the chance of destroying assignments which should have been used would be smaller when cleaning up less often. This solution would however use more memory, because of the time stamp on every assignment.

The stack overflow is not the only problem when dealing with LagNa assignments in small step. Because it is possible to use global variables in LagNa, problems can occur if a user tries to assign a global variable to a new value and the program gets interrupted by an event during this assignment. This would mean that the global variable would not contain the value intended and therefor problems when using these variables later in the code could happen. Listing 4.16 shows one of the problem events can have when using global variables.

Listing 4.16: An example of globalvariable assignment (the ¤ indicate where the code gets interrupted by an event)

```
real GlobalVariable¤:=10;

run()
{
  GlobalVariable := GlobalVariable + 1;
}
```

If an event interrupts the code where the ¤ is (Listing 4.16), the event will be executed and after this the event will return to *run()* this means that “GlobalVariable:=10” has never been executed, so the GlobalVariable will never be initialized, and the program will generate an error when *run()* tries to add one to the value of GlobalVariable. This problem could be solved by defining that all declaration of public variables should be atomic, however this would require different set of semantic rules for describing declaration of global variables, besides the rules needed for local variable declaration. However this would not be possible because in operational semantics there are no difference on global and local variables. This is because global variables are only variables defined in the outermost scope. Should there be different rules for the outermost scope, it would also be logical to have different rules for the next most outermost scope and so on. However this would be very difficult, because it would then be necessary to know which scope every command belongs to in the semantic rules. For instance when a method returns a value it would be necessary to know which variable environment which should be updated.

Instead of solving this problem in the semantics, it could easily be solved at compiler time. This is because of the actions “Ignore” and “Unignore” which exist in the language LagNa. By using the action “Ignore” before defining any global variables and the action “Unignore” after defining all the global variables, event interrupts will not be a problem, because “Ignore” is made for the purpose of ignoring an event, and therefore initialization of a global variable will not be a problem.

Table 4.49 shows a summary of all the problems and solutions this text have described about small-step semantics.

Problem	**Normal Small-step**
Instant event	Not a problem.
Stack overflow	A problem, but possible to solve using garbage collecter
	Could also be solved by using atomic semantics for all commands, execpt composition.
	or by cleaning the program stack when leaving a scope.
	Or by making two rules for every commands, one if an event occur, and one if no event occur.
Interrupt of global variable assignment	A problem, which could be solved by using atomic semantics for assignment.
	Or by using Ignore - Unignore when declaration global variables

Table 4.49: Summary of the different problems and possible solutions for using big-step and small-step semantics for describing LagNa

Chapter 5

Program Verification

Program verification, is proving that an algorithm, in some language, actually does what is expected to do. This could for example be verifying that an algorithm written to calculate the factorial value of some number actually does that. A verification of such an algorithm would consist of some sort of theoretical proof. One way would be to make those directly based on ones semantics [NN99, page 169], but those can be very detailed and not very pratical to use for program verification. It could be nice if there were some way of going up an abstraction level and be able to carry out the verification proof there. In this chapter we will show two such systems, which captures the essential properties of commands and allows us to do program verification using those systems. We will argue for the choice of one of these systems and develop one for a subset of LagNa, and finally we will show that the developed system is actually applicable for doing program verification for algorithms written in LagNa.

5.1 Methods of Verification

In programming verification we have two main approaches [Bac86, page 82]:

1. Inference rules, developed by C. A. R. Hoare,
2. Predicate transformers, developed by E. W. Dijkstra.

Both of these approaches work with defining pre- and postconditions, that is, defining some property that must be present before a command is executed, the precondition, and a property that will hold after the command has been executed, the postcondition. For this purpose we will use predicates. The idea is that they are functions that, based on the state of the program, returns a truth value. Without going in to further details about how we will present the state of a program, we present the following informal definition of predicates:

Definition 5.1. Predicates
A predicate P is a function

$$P : \text{“}state\text{''} \rightarrow b,$$

where $b \in B = \{tt, ff\}$.

As an example consider an algorithm to interchange the value of two variables using a temp variable. Not considering the details of type declaration the code would be as seen in Listing 5.1.

Listing 5.1: Interchange of two variables

```
temp := a;
a := b;
b := temp;
```

The state of a program running that algorithm would then be dependent on the values of the program variables. To express pre- and postcondition for that program, we need a way to relate the initial and final values of the program variables to each other. We call those logical variables. Then, if we denote the initial value of a by a_i and the initial value of b by b_i, we can write our pre- and postconditions as:

$$\begin{aligned} \text{Precondition:} \quad & a = a_i \wedge b = b_i \\ \text{Postcondition:} \quad & a = b_i \wedge b = a_i \end{aligned}$$

In the following two sections we will look at how program verification would be done for our interchange algorithm in both the mentioned approaches to program verification.

5.1.1 Inference Rules

Inference systems works by setting up inference rules [Bac86, page 87]

Definition 5.2. Inference rule
If we have two predicates P, Q and a command C, then we define the notation

$$\{P\}C\{Q\},$$

to express that if the state of the program initially satisfies P, the precondition, and C is executed. Then, if C terminates, the final state of the program will satisfy Q, the postcondition.

Now for the purpose of using that to verify our previous code example we need to consider how inference rules for assignment and composition would look like. Consider first the assignment $x := e$. Now suppose we want to guarantee some postcondition Q. The precondition would then be the predicate we would get if we took Q and substituted x for e everywhere in that predicate. We will use the following notation, $Q[x \mapsto e]$, to express that x has been substituted with e everywhere in the predicate. The inference rule would then look like

$$\{Q[x \mapsto e]\}\{x := e\}\{Q\}.$$

There are some further complications with the assignment inference rule. First the type of x and e would have to match each other and the expression e would have to be well defined, for instance no division by zero can occur. Their formalization though would clutter our inference rules with more details than we are interested in, so we will proceed with the assumption that those situations will not occur. Secondly consider composition of two commands $C_1; C_2$, we will write the composition inference rule for this in the following way.

$$\frac{\{P\}C_1\{Q\}, \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

which means, that we assume executing C_1 with precondition P guarantees postcondition Q, and executing C_2 with precondition Q guarantees postcondition R, then executing $C_1; C_2$ with precondition P would guarantee postcondition R.

Now we can provide a proof of the interchange algorithm using our two inference rules, summarized in Table 5.1 on the following page.

Theorem 5.1.
$\{a = a_i \wedge b = b_i\}$
temp := a;
a := b;
b := temp;
$\{a = b_i \wedge b = a_i\}$

[assignment]	$\{ \text{Q } [x \mapsto e]\} x := e; \{Q\}$
[composition]	$\dfrac{\{P\}C_1\{Q\}, \{Q\}C_2\{R\}}{\{P\}C_1C_2\{R\}}$

Table 5.1: Example inference system

Proof. Using the [assignment] rule on b := temp and a := b gives us

$$\{a = b_i \wedge temp = a_i\} \text{ b := temp; } \{a = b_i \wedge b = a_i\} \tag{5.1}$$

and

$$\{b = b_i \wedge temp = a_i\} \text{ a := b } \{a = b_i \wedge temp = a_i\}. \tag{5.2}$$

Using the rule of composition [composition] on (5.1) and (5.2) gives us

$$\{b = b_i \wedge temp = a_i\} \text{ a := b; b := temp; } \{a = b_i \wedge b = a_i\}. \tag{5.3}$$

Using [assignment] on temp := a; then gives us that

$$\{b = b_i \wedge a = a_i\} \text{ temp := a } \{b = b_i \wedge temp = a_i\}. \tag{5.4}$$

Now, since $b = b_i \wedge a = a_i \Leftrightarrow a = a_i \wedge b = b_i$, using the rule of composition on (5.3) and (5.4) this gives us the desired proof. □

5.1.2 Predicate transformers

For the predicate transformers we introduce the weakest precondition, with the following definition taken from [Bac86, Page 88]:

Definition 5.3.
If Q is a predicate and S a statement then the weakest precondition of S with respect to Q, denoted wp(S,Q), is a predicate describing the set of all initial states, such that execution of S begun in any one of the states is guaranteed to terminate in a state satisfying Q.

To illustrate the use of the predicate transformers we will prove the same example as we did with the inference systems. For this purpose, we first have to introduce the needed rules in Table 5.2 on the next page, the idea behind these rules are the same as we used for our inference system.

Then we use our weakest precondition rules to prove the interchange algorithm.

[assignment]	$wp(x := e, P) \equiv P[x \mapsto e]$
[composition]	$wp(S_1; S_2, Q) \equiv wp(S_1, wp(S_2, Q))$

Table 5.2: Example WP system

Theorem 5.2.
$wp(\text{temp := a; a := b; b := temp; }, a = b_i \wedge b = a_i) \equiv a = a_i \wedge b = b_i$

Proof. Using the [assignment] rule on b := temp and a:= b gives us

$$wp(\text{b := temp }, a = b_i \wedge b = a_i) \equiv a = b_i \wedge temp = a_i \tag{5.5}$$

and

$$wp(\text{a := b }, a = b_i \wedge temp = a_i) \equiv b = b_i \wedge temp = a_i. \tag{5.6}$$

Using the rule of composition, [composition], on (5.5) and (5.6) gives us

$$wp(\text{a := b; b := temp; }, a = b_i \wedge b = a_i) \equiv b = b_i \wedge temp = a_i. \tag{5.7}$$

Now using [assignment] on temp := a we have

$$wp(\text{temp := a }, b = b_i \wedge temp = a_i) \equiv b = b_i \wedge a = a_i. \tag{5.8}$$

Now, since $b = b_i \wedge a = a_i \Leftrightarrow a = a_i \wedge b = b_i$, using the rule of composition on (5.7) and (5.8) gives us the desired proof. □

5.1.3 Choice of method

Looking at the two proofs for our interchange algorithm it is clear that they look alike, but there is one significant difference. Hoares approach says that if a command terminates then it will satisfy the postcondition from a given precondition. This is called *partial correctness*. In Djikstra's method we write about the set of all preconditions that will guarantee *termination* in a state satisfying the postcondition [Bac86, page 124].

partial correctness + termination = total correctness

Since bots, programmed in our language, is event driven, we can never be certain that a specific command is terminated Section 4.4 on page 43. Therefore we can only describe the behavior **if** a command terminates, so we will not prove total correctness. Therefore we will use Hoares approach.

[assignment]	$\{ \text{ P } [x \mapsto e\ ]\}\ x := e;\ \{ \text{ P } \}$
[composition]	$\dfrac{\{P\}C_1\{Q\}, \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$
[consequence]	$\frac{\{P'\}C\{Q'\}}{\{P\}C\{Q\}}\quad if P \Rightarrow P'\ and\ Q' \Rightarrow Q$
[if]	$\frac{\{\mathcal{B}[b]\wedge P\}C_1\{Q\},\{\neg\mathcal{B}[b]\wedge P\}C_2\{Q\}}{\{P\}\text{if } b\ \underline{\{}C_1\underline{\}}\text{ else }\underline{\{}C_2\underline{\}}\{Q\}}$
[while]	$\frac{\{\mathcal{B}[b]\wedge P\}C\{P\}}{\{P\}\text{while } b\ \underline{\{}C\underline{\}}\{\neg\mathcal{B}[b]\wedge P\}}$

Table 5.3: LagNa inference system

5.2 Inference System

Using the notation introduced in section 5.1.1, we will set up an inference system, for LagNa. This can be seen in Table 5.3.

The [assignment] and [compsition] are just as we saw them earlier when we introduced the inference systems.

Next we have the rule [consequence], rule of consequence, that says we can strengthen our precondition and weaken our postcondition.

The [if] says that if we execute $if\ b\ \{C_1\}\ else\ \{C_2\}$ from a situation where the precondition P holds, and the command terminates, then we have that the postcondition Q holds if we can deduce that

- if C_1 is executed from a state where $\mathcal{B}[b] \wedge P$ hold and terminates, then Q holds on the final state
- if C_2 is executed from a state where $\neg\mathcal{B}[b] \wedge P$ hold and terminates, then Q holds on the final state

The idea behind the [while] rule is that the predicate P can be thought of as the *invariant* of the while statement. That is, P must hold before and after each execution of C. Additionally the command says that b will no longer be true when the execution of while terminates.

5.3 Provability of Assertions

Just as it is possible to make derivation trees using transition rules in operational semantics [Hüt04, page 41], it can be done with inference rules, we will call that inference trees. In Appendix F on page 132 we see the inference tree corresponding to the proof we made for our interchange algorithm using inference rules.

An inference tree shows how we can use the rules in our inference system to show a specific assertions holds, we will use that in a definition of provability in our inference system.

Definition 5.4. Provability
We say that a partial correctness assertion, $\{P\}S\{Q\}$, is proved in our inference system, if we can make an inference tree with $\{P\}S\{Q\}$ in the root. We shall write this as:

$$\vdash \{P\}S\{Q\}$$

Before continuing we have to redefine predicates as we saw them earlier to accommodate the fact that our semantics use the environment-store model, Section 4.2.1 on page 41, and not "states" as in the earlier definition.

Definition 5.5. Predicates
A predicate P is a function

$$P : \mathit{Envl} \times \mathit{Store} \to b,$$

where $b \in B = \{\mathit{tt}, \mathit{ff}\}$.

We will now show the relationship between our operational semantics and our inference system. We will prove, that if some partial correctness property can be proven for the inference system, then it holds for the operational semantics (soundness), and if some partial correctness property holds for the operational semantics then we can give a proof for it using the inference system (completeness).

Before doing so, we will give a definition of *validity* of a partial correctness assertion.

Definition 5.6.
A partial correctness assertion, $\{P\}S\{Q\}$, is said to be valid, if $\forall(envl, sto) \in Envl \times Store$, we have, that if

$$P(envl, sto) = tt$$

and

$$\exists(envl', sto') \in Envl \times Store$$

such that

$$\langle S, (envl, sto)\rangle \rightarrow (envl', sto')$$

then

$$Q(envl', sto') = tt.$$

We will write this as:

$$\models \{P\}S\{Q\}.$$

We can then express the soundness property as

$$\vdash \{P\}C\{Q\} \Rightarrow \models \{P\}C\{Q\}$$

and the completeness as.

$$\models \{P\}C\{Q\} \Rightarrow \vdash \{P\}C\{Q\}$$

The chosen strategies in the proofs for soundness and completeness has been inspired by [NN99, page 184-190].

Lemma 5.1. ***Soundness***
For every partial correctness assertion $\{P\}C\{Q\}$ of the inference system in Table 5.3 we have

$$\vdash \{P\}C\{Q\} \Rightarrow \models \{P\}C\{Q\}.$$

Proof. **Inference rule [assignment]:** Assume that

$$\langle x := e, envl, sto\rangle \rightarrow envl, sto'$$

and

$$P[x \mapsto e](envl, sto) = tt$$

We then need to prove that $P(envl, sto') = tt$. From the assignment axioms of Table 4.15 on page 59 we have that the location of x has been updated with the value of e. So every instance of x has been substituted with e, we write this as $(envl, sto') = (envl, sto)[x \mapsto e]$. Now since $P[x \mapsto e](envl, sto) = tt$ we get that $P((envl, sto)[x \mapsto e])$, thus $P(envl, sto') = tt$ as needed.

Inference rule [compsition]: Assume that

$$\models \{P\}C_1\{Q\} \wedge \models \{Q\}C_2\{R\}$$

we then have to prove

$$\models \{P\}C_1; C_2\{R\}$$

Let $(envl, sto)$ and $(envl'', sto'') \in Envl \times Store$ be such that $P(envl, sto) = tt$ and $\langle C_1; C_2, envl, sto\rangle \rightarrow (envl'', sto'')$, then from the transition rules for the composition command in Table 4.18 on page 61, we have that $\exists(envl', sto') \in Envl \times Store$ such that

$$\langle C_1, envl, sto\rangle \rightarrow envl', sto' \wedge \langle C_2, envl', sto'\rangle \rightarrow envl'', sto''$$

Then $\models \{P\}C_1\{Q\}$ gives us that $Q(envl', sto') = tt$ which together with $\langle C_2, envl', sto'\rangle \rightarrow envl'', sto''$ and $\models \{P\}C_2\{Q\}$ gives us what we needed, that $R(envl'', sto'') = tt$

Inference rule [consequence]: Assume that there exists predicates P' and Q' such that

$$\models \{P'\}C\{Q'\} \wedge P \Rightarrow P' \wedge Q' \Rightarrow Q.$$

Now consider $envl, sto$ and $envl', sto' \in Envl \times Store$ such that

$$P(envl, sto) = tt \wedge \langle C, envl, sto\rangle \rightarrow envl', sto'.$$

We then have that $P'(envl, sto) = tt$ and $\models \{P'\}C\{Q'\}$ then gives us that $Q'(envl', sto') = tt$, and since $Q' \Rightarrow Q$ we have that $Q(envl', sto') = tt$ as we wanted.

Inference rule [if]: Here we assume that

$$\models \{\mathcal{B}[b] \wedge P\}C_1\{Q\} \wedge \models \{\neg\mathcal{B}[b] \wedge P\}C_2\{Q\}$$

Now consider $envl, sto$ and $envl', sto' \in Envl \times Store$ such that $P(envl, sto) = tt$ and

$$\langle \text{if } b\ \{C_1\} \text{ else } \{C_2\}, envl, sto\rangle \rightarrow (envl', sto')$$

The remainder of the proof is split into the cases where $\mathcal{B}[b] = tt$ and $\mathcal{B}[b] = ff$. We show the remainder for the case where it is tt, the other case is proven in a similar way. Now we have that $(\mathcal{B}[b] \wedge P(envl, sto)) = tt$, and from the transition rules for the *if* command in Table 4.22 on page 65 we get

$$\langle \text{if } b\ \{C_1\} \text{ else } \{C_2\}, envl, sto\rangle \Rightarrow \langle \text{localscope } C_1 \text{ end}, envl, sto\rangle,$$

then the transition rules for commands in a scope in Table 4.19 on page 62 give us that $\exists envl', sto' \in Envl \times Store$ such that

$$\langle \text{localscope } C_1 \text{ end}, envl, sto\rangle \rightarrow envl', sto'.$$

Now from $\models \{\mathcal{B}[b] \wedge P\} C_1 \{Q\}$ we get what we wanted, that $Q(envl', sto') = tt$.

Inference rule [while]: We wish to prove

$$\models \{P\}\text{while } b \ \{C\}\{\neg\mathcal{B}[b] \wedge P\}$$

and for that we assume

$$\models \{\mathcal{B}[b] \wedge P\}\text{localscope } C \text{ end;}\{P\},$$

and consider $envl, sto$ and $envl'', sto'' \in Envl, Store$ such that

$$P(envl, sto) = tt \wedge \langle \text{while } b \ \{C\}, envl, sto\rangle \rightarrow envl'', sto'',$$

we then need to prove that $\neg\mathcal{B}[b] \wedge P(envl'', sto'') = tt$. By the transitionrule for while in Table 4.23 on page 66 we get that:

$$\langle \text{while } b \ \{C\}, envl, sto\rangle \Rightarrow \langle \text{if } b \ \{\text{localscope } C \text{ end; while } b \ \{C\}, envl, sto\rangle$$

Now we have two cases, either $\mathcal{B}[b] = tt \vee \mathcal{B}[b] = ff$. We start with the latter case. Then by the semantics of the *if* rule it follows directly from the transition rule [if-2E], in semantic at Table 4.21 on page 64, that $(envl'', sto'') = (envl, sto)$ and it is clear that $\neg\mathcal{B}[b] \wedge P(envl'', sto'') = tt$ as required. Finally we take the case where $\mathcal{B}[b] = tt$ and use induction on the shape of the derivation tree in the semantics. This amounts to proving that the property holds for all the axioms in the transition system. Assume that it holds for all composite derivation trees, the induction hypothesis, and then prove that it holds for the conclusions [NN99, page 28]. The base case follows trivially, and then we have

$$\langle \text{localscope } C \text{ end; while } b \ \{C\}, envl, sto\rangle$$

and composition gives us that $\exists\, (envl', sto') \in Envl \times Store$ such that

$$\begin{aligned}&\langle \text{localscope } C \text{ end;}, envl, sto\rangle \rightarrow (envl', sto') \wedge\\ &\qquad \langle \text{while } b \ \{C\}, envl', sto'\rangle \rightarrow (envl'', sto'')\end{aligned}$$

This gives us that $\mathcal{B}[b] \wedge P(envl, sto) = tt$ and together with our original assumption $\models \{\mathcal{B}[b] \wedge P\}C\{P\}$, we have that $P(envl', sto') = tt$. Now the induction hypothesis applied to $\langle \text{while } b \ \{C\}, envl', sto'\rangle \rightarrow (envl'', sto'')$ gives that $\neg\mathcal{B}[b] \wedge P(envl'', sto'') = tt$ and completes our proof. $\square$

Before we turn to the proof of completeness we introduce a new predicate

Definition 5.7. Weakest Liberal Precondition
For all commands C and predicates Q, we say that

$$wlp(C, Q)(envl, sto) = tt$$

if and only if for all environment-stores $(envl', sto') \in Envl \times Store$ we have that,

$$\langle C, (envl, sto)\rangle \rightarrow (envl', sto') \Rightarrow Q(envl', sto') = tt$$

The following lemma shows that $wlp(C, Q)$ is the weakest possible precondition for C and Q.

Lemma 5.2.
For all commands C and predicate Q we have that

$$\models \{wlp(C, Q)\}C\{Q\} \tag{5.9}$$

and

$$\text{if} \models \{P\}C\{Q\} \text{then } C \Rightarrow wlp(C, Q) \tag{5.10}$$

Proof. Let $(envl, sto)$ and $(envl', sto') \in Envl \times Store$ be such that $\langle C, (env, sto)\rangle \rightarrow (env', sto')$ and $wlp(C, Q)(envl, sto) = tt$. By Definition 5.7 of weakest liberal precondition we get that $Q(envl', sto') = tt$, which proves (5.9). Second, assume $\models \{P\}C\{Q\}$, and let $P(envl, sto) = tt$. If $\langle C, (envl, sto)\rangle \rightarrow (envl', sto')$ then $Q(envl', sto') = tt$ which gives us the wanted, that $wlp(C, Q)(envl, sto) = tt$, and (5.10) is proved. □

Lemma 5.3. ***Completeness***
For every partial correctness assertion $\{P\}C\{Q\}$ of the inference system in Table 5.3 we have

$$\models \{P\}C\{Q\} \Rightarrow \vdash \{P\}C\{Q\}$$

Proof. We assume $\models \{P\}C\{Q\}$, and need to prove that $\vdash \{P\}C\{Q\}$. But Lemma 5.2 and rule [consequence] of the inference system gives us that we only have to show

$$\vdash \{wlp(C, Q)\}C\{Q\}. \tag{5.11}$$

The proof of that is split into separate parts for each of the commands. They will be proved by induction on the statement C. The idea is to prove that the assertion holds for all the basis elements, which follows directly by the rules in the inference tree. Then Assume that the property holds for all the immediate components, the induction hypothesis, and then prove that it holds for the element it self [NN99, page 21].

The statement $[x := e]$**:** From the semantics of assignment it is easy to verify that

$$wlp(x := e, Q) = Q[x \mapsto e],$$

and then the result follows directly be the inference rule [assign].

The statement $[C_1; C_2]$**:** The induction hypothesis gives us that

$$\vdash \{wlp(C_2, Q)\}C_2\{Q\} \wedge \vdash \{wlp(C_1, wlp(C_2, Q))\}C_1\{wlp(C_2, Q)\}$$

The composition inference rule then gives

$$\models \{wlp(C_1, wlp(C_2, Q))\}C_1; C_2\{Q\},$$

and we need to prove that

$$wlp(C_1; C_2, Q) \Rightarrow wlp(C_1, wlp(C_2, Q)),$$

since the rule of consequence would then give us our proof. So we assume that $wlp(C_1; C_2, Q)(env, store) = tt$, and need to show that $wlp(C_1, wlp(C_2, Q))(env, store) = tt$. This follows trivially unless $\exists(envl', sto') \in Envl \times Store$ such that $\langle C_1, (envl, sto)\rangle \rightarrow (envl', sto')$. We must then prove that $wlp(C_2, Q)(envl', sto') = tt$. Again this follows trivially unless $\exists(envl'', sto'') \in Envl \times Store$ such that $\langle C_2, (envl', sto')\rangle \rightarrow (envl'', sto'')$, and then we must prove $Q(envl'', sto'') = tt$. But from the inference rule of composition, [compsition] we know that $\langle C_1; C_2, (envl, sto)\rangle \rightarrow (envl'', sto'')$, so $Q(envl'', sto'') = tt$ follows from $wlp(C_1; C_2, Q)(envl, sto) = tt$.

The statement [if b $\{C_1\}$ else $\{C_2\}$]**:** Applying the induction hypothesis to C_1 and C_2 we get that

$$\models \{wlp(C_1, Q)\}C_1\{Q\} \wedge \models \{wlp(C_2, Q)\}C_2\{Q\}.$$

Now define the predicate P by

$$P = (\mathcal{B}]b] \wedge wlp(C_1, Q)) \vee (\neg\mathcal{B}[b] \wedge wlp(C_2, Q)).$$

Then we get that

$$(\mathcal{B}[b] \wedge P) \Rightarrow wlp(C_1, Q)$$

and

$$(\neg\mathcal{B}[b] \wedge P) \Rightarrow wlp(C_2, Q),$$

applying the rule of consequence, [consequence], then gives us

$$\models \{\mathcal{B}[b] \wedge P)\}C_1\{Q\} \wedge \models \{\neg\mathcal{B}[b] \wedge P\}C_2\{Q\}.$$

This allows us to use the rule of inference for if, [if], and we get

$$\models \{P\}\text{if } b\ \{C_2\} \text{ else } \{C_2\}\{Q\}.$$

This is actually the desired result, which can be seen by showing that

$$wlp(\text{if } b\ \{C_2\} \text{ else } \{C_2\}, Q) \Rightarrow P$$

which is straightforward by looking at the cases of b.

The statement [while b $\{C\}$]**:** Define the predicate P by

$$P = wlp(\text{while } b\ \{C\}, Q).$$

We start by showing that

$$(\neg\mathcal{B}[b] \wedge P) \Rightarrow Q \tag{5.12}$$

Let $(envl, sto) \in Envl \times Store$ be such that $\neg\mathcal{B}[b] \wedge P(envl, sto) = tt$. Then, by $\langle \text{while } b\ \{C\}, (envl, sto)\rangle \rightarrow$ we have that $Q(envl, sto) = tt$. Next we show that

$$(\mathcal{B}[b] \wedge P) \Rightarrow wlp(C, P). \tag{5.13}$$

Now to verify that, let $(envl, sto) \in Envl \times Store$ be such that $\mathcal{B}[b] \wedge P(envl, sto) = tt$. The implication is then obvious unless $\exists(envl', sto') \in Envl \times Store$ such that $\langle C, (envl, sto)\rangle \rightarrow (envl', sto')$. Then we have to prove that $P(envl', sto') = tt$. We now have two situations. First we assume that $\exists(envl'', sto'') \in Envl \times Store$ such that $\langle \text{while } b\ \{C\}, (envl', sto')\rangle \rightarrow (envl'', sto'')$, then the semantics for while gives us that $\langle \text{while } b\ \{C\}, (envl, sto)\rangle \rightarrow (envl'', sto'')$, and now since $P(envl, sto) = tt$ we have that $Q(envl'', sto'') = tt$ from Lemma 5.2, but that again gives us that $P(envl', sto') = tt$ as we wanted. Now the second case we assume that there does not exists a $(envl'', sto'')$ such that $\langle \text{while } b\ \{C\}, (envl', sto')\rangle \rightarrow (envl'', sto'')$, but then $P(envl', sto') = tt$ follows directly.

Now apply the induction hypothesis to C and we get that

$$\models \{wlp(C, P)\}C\{P\},$$

and together with (5.13) and the inference rule of consequence, [consequence], we get $\models \{\mathcal{B}[[b]] \wedge P\}C\{P\}$. Now applying the inference rule for while we get

$$\models \{P\}\text{while } b\ \{C\}\{\neg\mathcal{B}[b] \wedge P\}.$$

Now using (5.12) and the rule of consequence we get the desired result that

$$\models \{P\}\text{while } b\ \{C\}\{Q\}.$$

□

Theorem 5.3.
For every partial correctness assertion $\{P\}C\{Q\}$ of the inference system in Table 5.3 we have

$$\models \{P\}C\{Q\} \Leftrightarrow \vdash \{P\}C\{Q\}$$

Proof. The proof follows directly from Lemma 5.1 and 5.3. □

Now having proved Theorem 5.3, we know that any properties, which can be proven with our inference system, actually holds in our semantics (Soundness). Also properties for the shown subset of our semantics which holds, can be proved in our inference system (Completeness).

5.4 Limitations of program verification

Program verification, the act of being able to say that you *know* that a certain program will do something correctly is a very aspiring concept. More and more often we see computer programs being used in mission critical situations where it is nothing short of a disaster if the program does something wrong. To name a few examples we have the programs controlling the robots that we send to Mars, or military diagnostics software that is responsible for informing if a nuclear strike is imminent. In both cases a program failure is critical. Mars is so far away that a project could potentially be a complete waste if an unrecoverable error were to occur, and likewise it would be disastrous if the military where to decide to make a nuclear retaliation strike based on incorrect information. But would a verification of those programs allows us to sleep better at night knowing that the program is "correct"? Brian Cantwell Smith says that "The short answers is no" in an article in [CFR93, page 276], and goes on to mention problems like programs with human interaction or the sheer complexity of verification. State of the art in program verification as of 1993 extended to verifying programs with line counts in the hundreds [CFR93, page 277] not many programs in wide use around the world today can be written with such few lines of codes, and is very far from the amount of code required for example in an operating system. Another problem is that the concept of verifying a program works, from a situation where the program executes on a "perfect virtual machine". A program verified from that assumption never takes into account problems in the code at a lower level, for instance like the operating system. Or even if the verified program is running at the lowest level we still have the possibility of hardware failure. Program verification is a very interesting and useful theoretical practice in Computer Science, but, as it has been shown, it is very important to realise its pratical limitations.

Chapter 6

The Compiler

In this chapter we will describe the three common phases of compiler construction and how we have implemented the compiler for the language LagNa.

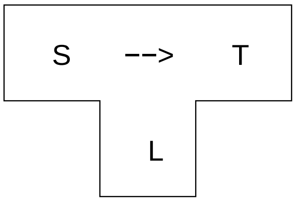
6.1 Tombstone diagrams

Tombstone diagrams are a tool, that can be used to figure out how a compiler should be implemented in order to obtain the compilers system platform.

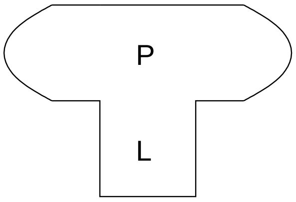
Tombstone diagrams consist of four elements: a program written in a language, a translator written in a language, an interpreter written in a language and a machine implemented in hardware[WB00]. The four elements are shown on Figure 6.1 on the next page.

The game engine is running on the Java Virtual Platform (JVM), so we want our compiler to produce code that can run on the JVM. This is shown on Figure 6.2 on the following page.

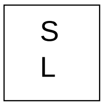
This looks fairly simple to do, but since our language must be considered a high level language, there is a fairly big task hidden in this diagram. Although we only have to create and implement the 'T' shaped tombstone, there is a problem since we will have to translate our high level language into a low level language such as the JVM. This can be done; the JVM is very well documented, but it is also a large piece of software with alot of instructions. Therefore we have chosen to add another element in the diagram and reuse the work of the Java authors. We will simply compile our language into java instead, thus adding an element of translating Java to JVM which is done using the Java compiler. However we have thought about compiling directly to the JVM, and implemented a function just to show how it could be done. The code for the mapping of the function is shown on Figure 6.5. The tombstone diagram for the solution we have chosen is portrayed on Figure



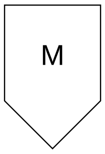
(a) Translator written in L translating S into T.



(b) Program P written in L.



(c) Interpreter interpreting S expressed in L.



(d) Machine M.

Figure 6.1: The tombstone elements.

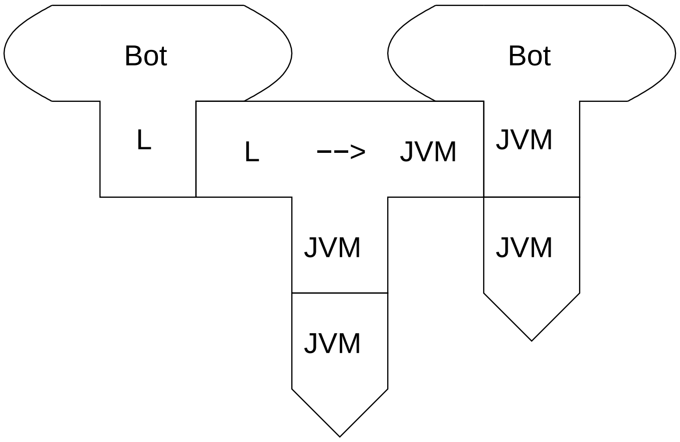


Figure 6.2: LagNa (L) to JVM.

6.3.

This extra element reduces the work we have to do significantly, since we are all familiar with Java. The choice of using the Java compiler also brings some advantages, such as optimization of code, which is build into the Java compiler [javb]. However it leaves us with less control.

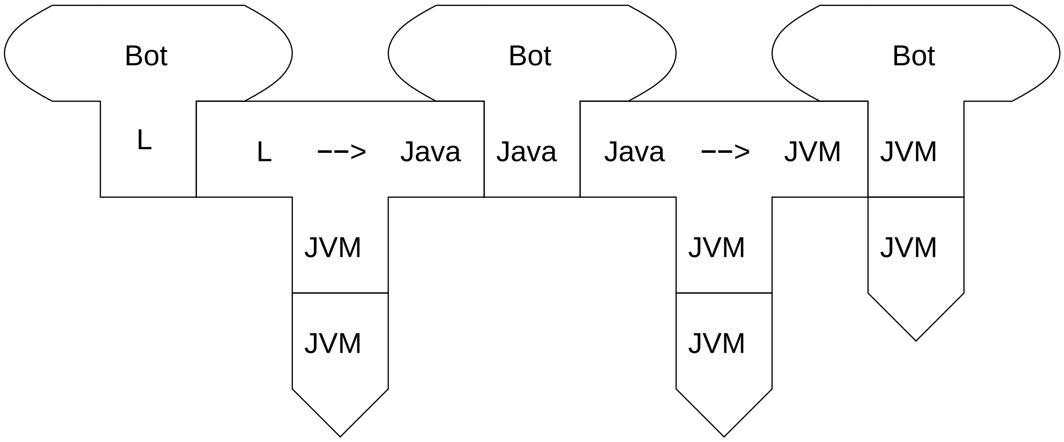


Figure 6.3: LagNa (L) to Java.

There is a third and alternative way of translating the bots: to convert the engine to an abstract machine interpreter, and just feed the engine with bots coded in LagNa directly. The diagram for this is shown on Figure 6.4.

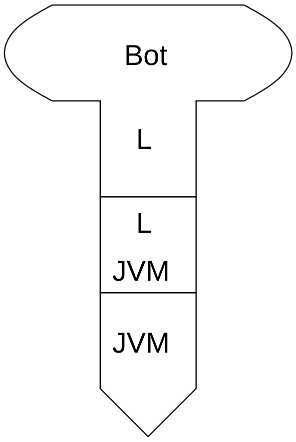


Figure 6.4: The abstract machine interpreting LagNa (L).

This would mean merging the game engine and the interpreter. To make an abstract machine would required to create a language for it. Thus to make an easy transition from the LagNa language to the abstract machine, issues would have to be discussed in the language design phase. e.g. how easy should the language map to the abstract machine language.

Normally the building of a compiler can be divided into three phases: the syntactic analysis phase (or syntax analysis), contextual analysis and the code generation phase.

Syntax analysis This phase has the purpose of converting the source code of a program to an abstract syntax tree. This means making lexical analysis and generating tokens that can be used to do the syntactical verification of the program. This phase is closely tied to the development of the language grammar. The product of this phase should, if successful, be a parser that will verify the syntax of a program written in the language the parser is generated to recognize. It is often wise for the parser to generate an abstract syntax tree, since this makes the code generation phase easier. Because all of the checking has been done before the code generation phase, so it just need to translate the code. The compiler only need to traverse the tree once, when it has to generate the code. The syntax analysis phase can be further divided into two smaller phases: scanning and parsing. The scanner only scans the source code and generates tokens that the parser can use to do syntax verification and construction of the abstract syntax tree.

Context analysis This phase has the purpose of dealing with all the constraints a language may have, which can not be implemented in the syntactical phase. As an example the constraint that all variables has to be declared before they are used, is such a constraint that would be implemented in this phase. This means that this phase of the compiler adds information to the abstract syntax tree generated by the syntactical phase. This is called decoration of the abstract syntax tree.

Code generation This phase "simply" has the task of actually generating the code that will do what the source program describes. The code generated in this phase is often referred to as object code. As an example in Java that would be the *.class* files, and in C it would be the *.o* files. In the case of the LagNa language it will be .java files.

6.2 Syntax analysis

For the generation of compiler we have used a tool, that aids in the development of the compiler, more specific a complete parser unit and the framework for the compiler. SableCC [Gag] is the name of this tool, and it generates everything that the compiler will have to do in the syntactical analysis phase from a grammar file. It will generate the scanner (or *lexer*) that reads and extracts tokens from the source code. It generates the parser that does the actual syntactical verification of the tokens. It checks if the source code can compile with the syntax of the language. Furthermore it generates an analyzer, which can be extended and used in the contextual analysis phase.

Listing 6.1: The grammar for LagNa that complies to the SableCC standard.

```
bot =
type_decl* bot_element* bot_run?;

bot_run =
P.run bot_element*;
```

The SableCC parser is a recursive decent parser which means it traverses the abstract syntax tree top down.

Part of the grammar for SableCC that is derived from our context free grammar is shown on Listing 6.1. The complete grammar is shown in appendix G.

SableCC generates nodes for each production in the grammar it is supplied with. These nodes are used by the parser, to generate the actual abstract syntax tree, from the source code, which is supplied to the parser.

6.3 Contextual analysis

The analyzer mentioned above is used in this phase of the compiler. We have extended the analyzer to handle topics like type checking and other constraints. We have chosen to have several constraints in our language. The first constraint relates to the declaration of variables, a variable has to be declared before it is used. This rule is a common rule in many languages, and it is a rule that is necessary if one wants to implement a single pass compiler. However we are not making a single pass compiler, and the reason for the rule is that we believe it makes it easier to read the code, thus giving a better overview of the source.

The reason that we are making a multi pass compiler is, that the programmer is not required to declare methods before they are used. The only way to check that, the method the programmer is calling, is declared somewhere in the code, is to do multi parsing.

Another constraint on the language is the presence of scope rules. There exist several combinations of scope rules. We have chosen what we found to be the most intuitive, which is static scope rules for everything; variables, methods and events. Static scope refers to the fact that the scope of a variable can be determined statically prior to execution [Seb04][page 211-213].

Another important aspect of language construction, is type checking. We have choosen strong typing, which means that the programmer cannot redefine variables after they have been defined. In this way the types can be resolved at compile time as well as at runtime [Seb04][page 206-208].

Type checking can be divided into two categories; static type checking and dynamic type checking. Static type checking refers to the fact that all type checking is done at compile time and dynamic type checking is done at runtime thus leaving some overhead compared to static. In earlier chapters we have describes the semantics of our language and the type rules. We have chosen to use static type checking.

The analyzer, when done, will have added extra information to the abstract syntax tree, e.g. types, making it a decorated syntax tree. This is the tree used in the code generation phase.

When done with this phase, we will have an abstract syntax tree, with information about types, and checking for declarations.

6.4 Code generation

In this phase we traverse the decorated abstract syntax tree to generate the code, and then the compilation is complete.

We have chosen that our compiler should generate Java code from LagNa source code. We thought about compiling to Java Byte Code (JBC), but as we have focus on the Syntax and Semantics of the language rather than the compiler we figured we would save some time compiling to Java instead. However to show that our language can be compiled to JBC we have included an example, of a bot written in LagNa (Listing 6.2) and the equivalent bot translated to JBC (Listing 6.3, Listing 6.4 and Listing 6.5).

```
run() {
  lookleft();
  shoot();
}

event OnStart() {
  real ret;
  ret := fac(3);
}

method real fac(real number) {
  real tmp := 1;
  while number <> 1 {
    tmp := tmp * number;
    number := number - 1;
  }
  return tmp;
}
```

Listing 6.2: Example of a bot

```
0: aload_0
1: invokevirtual #2; // lookleft()
2: aload_0
3: invokevirtual #3; // shoot()
4: return
```

Listing 6.3: Line 1 to line 4 from listing 6.2 in JVM code

```
0: aload_0
1: ldc2_w #4; // push double 3.0 from constant pool
2: invokevirtual #6; // fac()
3: dstore_1
4: return
```

Listing 6.4: Line 6 to line 9 from listing 6.2 in JVM code

To compile the bot to JBC we will first have to be aware of the class file format[LY99a]. We will not create the entire class file in this example, only translate the actual code shown. This means we pretend that the entire class file structure has been created, and that all the constant pool information has been filled. The constant pool can be viewed as a collection of function names and constants, that can be referenced to through an index. This means that the symbolic references, used to refer to the information in the constant pool, are imaginary in this example, since we do not actually know how the pool is constructed. The layout of the constant pool will of course have to be constructed should we compile the bots to work in the engine.

Since we only have reals in LagNa, we can use the largest floating point type available in the Java Virtual Machine, which is double. This will not make the code more efficient, but for simplicity reasons this would be our first choice, since it gives the highest precision. This means that we can use the Java Virtual Machine instructions which operates on double whenever possible as a standard. We will follow standards and use // for comments in the Java Byte Code example. For translating source code to JBC we will have to be familiar with The Java Byte Code instruction set reference.[LY99b]

We can compile *run* to the code shown on Listing 6.3.

The event OnStart can be compiled to the code shown on Listing 6.4.

The method *fac* is shown on Listing 6.5.

Optimization could be made in the compiler, that could detect if the high precession is unused such as integer operations like in fac(3); the 3 might as well be an integer. Other optimizations can be made in this example like replace the fac operation in OnStart with the result 6, since the argument to the function dose not change in the life of the OnStart event. Since we are compiling to the java compiler, we benefit from the optimization that the Java

```
0: dconst_1 // push double constant 1.0 onto stack
1: dstore_3 // store double at top of stack into variable 3 (tmp
    := 1)
2: dload_1  // load local variable 1 (number)
3: dconst_1
4: dcmpl
5: ifeq 17 // if (number = 1) jump to 17 else (number <> 1)
8: dload_3 // load tmp onto stack
9: dload_1 // load number onto stack
10: dmul   // multiples the two top stack elements
11: dstore_3 // store result of multiplication in tmp
12: dload_1  // load number onto stack
13: dconst_1
14: dsub     // subtract the two top elements from the stack
15: dstore_1 // store subtraction result in tmp
16: goto 2
17: dload_3 // load tmp onto stack
18: dreturn // return result of tmp
```

Listing 6.5: Line 11 to line 18 from listing 6.2 in JVM code

compiler does. However if we were to compile to JBC. Optimization would become an issue, that could be dealt with in the compiler. It is important that the compiler makes semantic-preserved transformations, so the program still performs as expected when optimized.

Chapter 7

Conclusion

This chapter concludes upon the fulfillment of the problem formulation. To recall, it was: "The objective of this project is to develop such a higher level programming language, which we will call LagNa. A special purpose langauge used for the development of bots for the game GamNa [Tea]. The language should be developed using both a well-formed syntax and semantics. Furthermore we will look into the possibility of doing program verification and we will implement a compiler for the language."

We developed the language using context-free grammars and operational semantics. We developed a system for program verification for a subset of our language, and used SableCC to help us implement the compiler. For a more in-depth analysis of the work done we will look at how well the quality goals of Section 2.1 on page 7 have been fulfilled, omitting the ones we rated as irrelevant: Generality and Portability.

Simplicity (Critical) A large part of simplicity is, as we mentioned, the read- and write-ability of the language. The only way to really know if this criterion was acomplished would be to subject the language to intensive testing by potential users. But that was never the intention of this project, so we will give our own observations on the simplicity of the language.

To obtain the critical simplicity of the language, we have refrained from including simultaneous or parallel constructs, even though the behaviour of the bot when using certain actions is parallel. This might have made the language more complex to read and understand, which is a problem considering the quality goal of simplicity. For instance we have the example of a pice of code (shown in Listing 7.1), where we want a bot to move in a "back and forth" pattern, and at the same time rotate the line of sight.

In a normal programming language commands are executed and completed sequentially, which makes it relatively easy to read and understand

```
run() {
  move(5);
  move(-5);
  look(10);
}
```

Listing 7.1: Queue problem example.

the program, but the example demonstrate the problem with our language. Using this intuition, one would expect the bot to move five units forward and then five units backward, resulting in the same position as it started in, and thereafter look 10 degrees to the right. But in our language this is not the case, because bots has to be able to move and look at the same time. This means that the actual behavior of the bot will be that the *move*(5) and the *look*(10) are both executed at the same time, and when the *move*(5) is done, then *move*(−5) begins. This reduces the ability to read and understand the code easily, although it is fairly easy to write once an understanding of the queue system has been achieved.

Data Types and Structures (Less Important) We have chosen to have three different basic data types in our languages: real, boolean and string. The reason why we only have three basic data types is to make the readability as simple as possible, in that we do not have large number of data types that resembles each other. To be able to create containers of one of these data types, we have created ID-lists. In our language we made the decision of not including the possiblity of making user defined data types. This limits the writeability in cases where for example the user wishes to make a method return more than one primitive data type.

Abstraction (Important) In LagNa we have the ability of abstraction in the form of the possibility to put code in methods and reuse it. This provides only a limited amount of abstraction to the language. To increase the possibilities of abstraction, we could introduce either the concept of a user defined data type or more predefined data types. This however would compromise our simplicity quality goal.

Cost (Important) In the cost criteria, program creation and program maintenance is heavily influenced by how easy it is to read and write programs in LagNa. This has been our primary concern throughout the development of our language. According to [Seb04] program creation is dependent on the write-ability of the language, and program maintenance primarily depends on readability. We believe from the conclusion of the other quality goals that our language, overall, is easy to read and write. Lastly we have the cost of program execution, but since the bots are run on a virtual machine, we are not concerned with execution time.

7.1 Future Works

The possible design flaw of the queue system can be corrected by the introduction of a form of construct, where it will be more clear that actions are executed simultaneously. In our case it would make sense to look at what the bot is composed of, namely a body and a head. We could then have made language constructs that represent the individual parts (body, head) of the bot and that it could be possible to command each part and make the execution parallel. The example could then be written as shown on Listing 7.2.

Listing 7.2: Parallel execution with head and body blocks

```
run() {
  body {
    move(5);
    move(-5);
  }
  head {
    look(10);
  }
}
```

With the introduction of the body and head parallel executed block constructs we have made the structure of the code and the intend more clear for the reader, but is it easier to understand? If one can understand the concept of simultaneous executing threads it is easier to understand since the body and head could represent a thread each and then the commands inside the block would execute like sequential code. However there are several problems with this construct. As an example, what happens if look is called inside both body and head?

Another alternative to the body and head construct is to extend the language with a *simultaneous* keyword that can be used to specify that the command associated with it should be executed at the same time. An example is shown on Listing 7.3

Listing 7.3: Parallel execution, with a simultanious keyword

```
run() {
  simultaneous {
    move(5);
    rotate(10);
  }
  move(-5);
}
```

Another way to make the language more sequential is to make hybrid commands that can complete the parallel tasks as shown on Listing 7.4

Listing 7.4: Parallel behavior with hybrid commands

```
run() {
  move_look(5, 10);
  move(-5);
}
```

This would make the program more sequential, but it would also introduce a fairly large amount of commands in the language in order to cover all combinations, so that would probably not be a better solution to improve simplicity. Another way to resolve this problem is to handicap bots to only be able to do one task at a time such as move and when that is finished then rotate. This would make the language more simple and execution of the bot would be sequential following the program very strictly thus making it easier to understand as well. But this handicap would prevent the bot programmer to achieve the full potential of the game and therefore is not a suitable solution.

Beside these problems with simultaneous behavior, we could have improved on the simplicity by simplifying our scope rules. This could be done by choosing a global scope like the monolithic scope, and inside this scope, new scope blocks could be created by letting the characters { and } denote a block. This way it would be possible to start new scopes everywhere, and anything outside the block would be global scope.

Another possibility could be to include the actual coding process in the game engine, that is, have an IDE, Integrated Development Environment, build into the actual game. This would also give the users easier and quicker access to actual testing of the programmed bots.

Appendix A

Actionstable

Action	Arguments	Description
ignore	list of events	Ignores the specified events.
unignore	list of events	Unignores the specified events.
rotate	real	Rotates the bot's body a number of degrees. A negative sign means counter clockwise, else clockwise
rotateleft		Rotates the bot's body counter clockwise.
rotateright		Rotates the bot's body clockwise.
look	real	Turns the bot's head a number of degrees. A negative sign means counter clockwise, else clockwise
lookleft		Turns the bot's head counter clockwise.
lookright		Turns the bot's head clockwise.
move	real	Moves forwards or backwards a specified distance. A negative sign indiates backwards.
moveforward		Makes the bot move forwards.
movebackward		Makes the bot move backwards.
strafe	real	Make the bot strafe left or right a specified distance. A negative sign indiates left.
strafeleft		Makes the bot strafe left.
straferight		Makes the bot strafe right.
chooseweapon	real	Selects a weapon to use when shooting. The argument is one of the following numbers: Number – Weapon 1 – pistol 2 – shotgun 3 – bombs 4 – rockets 5 – sniper 6 – minigun

Continues on the next page →

Action	Arguments	Description
shoot	real	Shoot with the selected weapon an optional number of times. If left out it will shoots continouesly
stop		Stops all actions.
stopmove		Stops any movement actions.
stoprotate		Stops any rotate actions.
stoplook		Stops any look actions.
stopshoot		Stops any shoot actions.
waitformove		Waits for all move actions to complete.
waitforrotate		Waits for all rotate actions to complete.
waitforlook		Waits for all look actions to complete.
waitforshoot		Waits for all shoot actions to complete.
waitforall		Waits for all actions to complete.
waitforevent		Waits for an event to occur.
gethealth		Returns the health of the bot.
getviewingangle		Returns the current angle the bot is looking at relative to movement.
canshoot		Returns a boolean to indicate if the bot can shoot at the moment, i.e. there are currently no firing delay set.
say	string	Makes the bot say the text specified in the argument. The text will only be displayed in approximately 5 seconds.

Table 7.1: Action table for the game engine.

Appendix B

Variables for LagNa

bot	run	bot-element
type-decl	type-decl-tail	identifier
assignment	assign-var	assign-list
types	var-name	var-char
number	digit	real
bool	ID-list-call	string
chars	char	uni-not-eol
uni-chars	exp	bool-exp
bool-exp-1	real-exp	real-exp-1
real-exp-3	value	var
event	event-name	id
event-list	event-list-name	method
method-tail	return-type	parameters
parameter	method-call	method-inputs
command	commands	action
action-name	if-exp	select-exp
cases	case	default
case-exp	case-exp-1	while-exp
special	comment	G-type-decl

Table 7.2: List of variables for LagNa

Appendix C

Tokens for LagNa

+	−	*	/
(	)	[	]
{	}	=	
,	<>	:	;
<	>	<=	>=
AND	OR	NOT	&
if	while	select	:=
event	method	run	random
sin	cos	power	sqrt
gethealth	gethviewingangle	canshoot	OnHit
OnDeath	OnCollision	OnSight	OnTargetHit
all	ignore	unignore	rotaterotateleft
rotateright	look	lookleft	lookright
strafe	strafeleft	straferight	move
moveforward	movebackward	chooseweapon	shoot
say	stop	stopmove	stoprotate
stoplook	stopshoot	waitforevent	waitforall
waitformove	waitforrotate	waitforlook	waitforshoot
default	real	string	boolean
true	false	UNICODE	[-] (0 .. 9)* [. (0 .. 9)*]
((a .. Z)((a .. Z)(0 .. 9))*)			

Table 7.3: List of tokens for LagNa

Appendix D

Derivation Tree

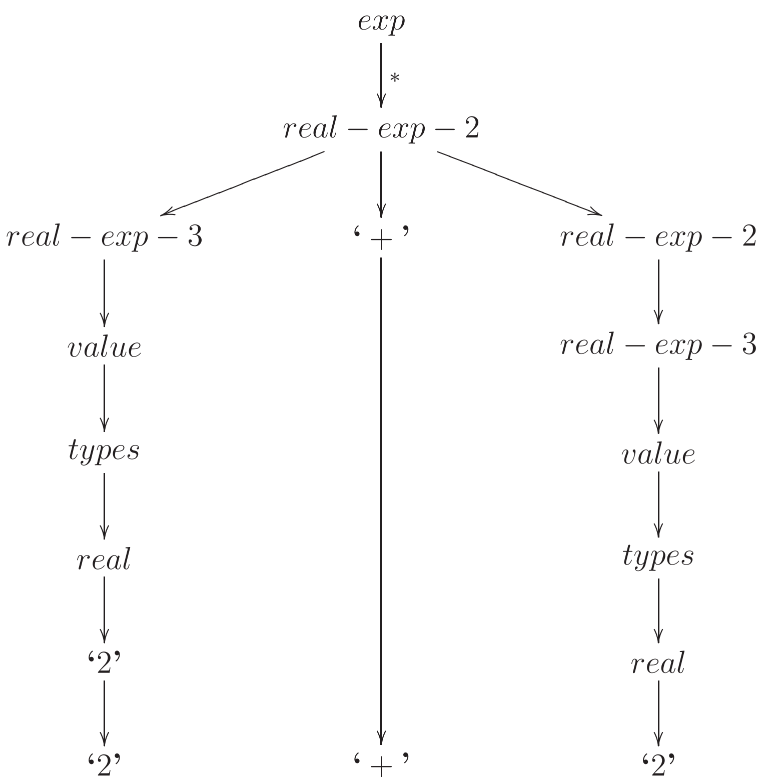


Figure 7.1: Derivation tree for the string "2+2" to show precedence. * means that we have skipped some steps.

Appendix E

Code Example

```
// It is like the main function in ANSI C, except it is being
    called again after event completion.
// This is where you code the default behavior of the bot.
run()
{
  moveforward();

  while true
  {
    look(360);
    waitforlook();
  }

}

method walk(real angle)
{
  rotate(angle);
  moveforward();
}

// The entry point for the bot. This is the first procedure call by
    the engine.
event OnStart()
{
  rotate(33);
}

// Our bot has collided with an object. So we change direction.
event OnCollision(real direction, real objecttype)
{
  real retning := random(-180,180);

  select retning
  {
    [-180:-90]
    {
      walk(-180);
    }
    ]-90:0]
    {
      walk(-90);
    }
    ]0:90]
    {
      walk(90);
    }
    ]90:180]
    {
      walk(180);
    }
  }
}

```

```
// We hit the target, so lets try again.
event OnTargetHit()
{
  shoot(1);
}

// We have spotted an object.
event OnSight(real distance, boolean teammate, real objecttype)
{
  // If the object is an enemybot
  if (objecttype = 3) AND (teammate = false)
  {
    // stop what we are doing
    stop();
    if distance <= 5
    {
      // the bot is close, so we choose a short range weapon (eg. shotgun)
      chooseweapon(2);
    }
    else
    {
      // the enemy is far away, so we choose a long range weapon (eg. sniper)
      chooseweapon(5);
    }
    shoot(1); //Shoot once
  }
}

// We are dead, but we have time for a death cry.
event OnDeath(string enemyname)
{
  string deathcry;

  deathcry := "Oh no i'am dead. Everybody shoot " & enemyname & " " & random(7,42) & " times.";

  say(deathcry);
}

// We have been shot, so we drop what we are doin
// to counter attack.
event OnHit(real retning)
{
  // we ignore all events
  ignore(all) ;
  stop();

  // turn to face our agressor
  look(retning);

  // choose a desperate weapon (eg. uzi)
  chooseweapon(6);

  // wait till we face enemy and attack
  sleep(retning);
  shoot();
}
```

Listing 7.5: An example of a program written in LagNa

Appendix F

Inference Tree

$$\frac{\{\text{a=a};\ \land\ \text{b=b};\}\ \text{temp := a};\ \{\text{temp=a};\ \land\ \text{b=b};\}, \quad \dfrac{\{\text{temp=a};\ \land\ \text{b=b};\}\ \text{a := b};\ \{\text{temp=a};\ \land\ \text{a=b};\},\{\text{temp=a};\ \land\ \text{a=b};\}\ \text{b := temp};\ \{\text{a=b};\ \land\ \text{b=a};\}}{\{\text{temp=a};\ \land\ \text{b=b};\}\ \text{a:=b};\ \text{b := temp};\ \{\text{a=b};\ \land\ \text{b=a};\}}}{\{\text{a=a};\ \land\ \text{b=b};\}\ \text{temp := a};\ \text{a := b};\ \text{b := temp};\ \{\text{a=b};\ \land\ \text{b=a};\}}$$

Appendix G

SableCC Grammer for LagNa

```
Helpers

        all = [0 .. 0xFFFF];
        digit = ['0' .. '9'];
        var_char = [['a' .. 'z'] + ['A' .. 'Z']];
        number = digit+;
        sign = '-';
        chars = [all - '"'];
        tab = 9;
        cr = 13;
        lf = 10;
        eol = cr lf | cr | lf;
        not_eol = [all - [cr + lf]];

Tokens

        lpar = '(';
        rpar = ')';
        ltub = '{';
        rtub = '}';
        lbra = '[';
        rbra = ']';
        colon = ':';
        semicolon = ';';
        dot = '.';
        coma = ',';
        plus = '+';
        minus = '-';
        multi = '*';
        div = '/';
        assign = ':=';
        bigger = '>';
        smaller = '<';
        biggerand = '>=';
        smallerand = '<=';
        equal = '=';
        notequal = '<>';
        and = 'AND';
        or = 'OR';
        not = 'NOT';
        stringop = '&';


        run = 'run';
        sin = 'sin';
        cos = 'cos';
        random = 'random';
        method = 'method';
        event = 'event';
        if = 'if';
        while = 'while';
        else = 'else';
        select = 'select';
        onhit = 'OnHit';
```

```
        ondeath = 'OnDeath';
        oncollision = 'OnCollision';
        onsight = 'OnSight';
        ontargethit = 'OnTargetHit';
        onstart = 'OnStart';
        event_all = 'all';
        var_real = 'real';
        var_bool = 'boolean';
        var_string = 'string';
        return = 'return';
        ignore = 'ignore';
        unignore = 'unignore';
        rotate = 'rotate';
        rotateleft = 'rotateleft';
        rotateright = 'rotateright';
        look = 'look';
        lookleft = 'lookleft';
        lookright = 'lookright';
        strafe = 'strafe';
        strafeleft = 'strafeleft';
        straferight = 'straferight';
        move = 'move';
        moveforward = 'moveforward';
        movebackward = 'movebackward';
        chooseweapon = 'chooseweapon';
        shoot = 'shoot';
        say = 'say';
        stop = 'stop';
        stopmove = 'stopmove';
        stoprotate = 'stoprotate';
        stoplook = 'stoplook';
        stopshoot = 'stopshoot';
        wait = 'wait';
        waitformove = 'waitformove';
        waitforrotate = 'waitforrotate';
        waitforlook = 'waitforlook';
        waitforshoot = 'waitforshoot';
        getviewingangle = 'getviewingangle';
        canshoot = 'canshoot';
        sleep = 'sleep';
        power = 'power';
        sqrt = 'sqrt';
        gethealth = 'gethealth';
        getview = 'getviewingangle';
        default = 'default';

        real = sign? number ( '.' number )?;
        bool = 'true' | 'false';
        string = '"' chars* '"';
        var_name = var_char (var_char | digit)*;

        blank = (' ' | tab | eol)+;
        comment = '//' not_eol* eol;

Ignored Tokens

        blank,
        comment;

Productions

        bot =
```

```
    g_type_decl* bot_element* bot_run?;

    bot_run =
    P.run bot_element*;

    run =
    T.run lpar rpar ltub commands rtub;

    bot_element =
    {bot_event}     P.event |
    {bot_method}    P.method;

    identifier =
    id list?;

    id =
    {real}          var_real |
    {string}        var_string |
    {bool}          var_bool;

    list =
    lbra rbra;

    type_decl =
    identifier var_name type_decl_assign? semicolon;

    g_type_decl =
    identifier var_name g_type_decl_assign? semicolon;

    type_decl_assign =
    assign exp;

    g_type_decl_assign =
    assign types;

    assignment =
    var_name id_list_call? assign exp semicolon;

    id_list_call =
    lbra exp rbra;

    exp =
    bool_exp string_exp_tail?;

    string_exp_tail =
    stringop exp;

    bool_exp =
    bool_exp_a bool_exp_tail?;

    bool_exp_tail =
    logic bool_exp;

    logic =
    {logic_and}     and |
    {logic_or}      or;

    bool_exp_a =
    not? real_exp;

    real_exp =
    real_exp_a real_exp_tail?;
```

```

real_exp_tail =
equal_op real_exp;

equal_op =
{equal_op_equal}        equal |
{equal_op_noequal}      notequal;

real_exp_a =
real_exp_b real_exp_a_tail?;

real_exp_a_tail =
bool_op real_exp_a;

bool_op =
{bool_op_bigger}        bigger |
{bool_op_smaller}       smaller |
{bool_op_biggerand}     biggerand |
{bool_op_smallerand}    smallerand;

real_exp_b =
real_exp_c real_exp_b_tail?;

real_exp_b_tail =
term real_exp_b;

term =
{term_plus}     plus |
{term_minus}    minus;

real_exp_c =
variable real_exp_c_tail?;

real_exp_c_tail =
factor real_exp_c;

factor =
{factor_multi}  multi |
{factor_div}    div;

variable =
{exp_variable}          types |
{exp_minus}             minus? variable_a;

variable_a =
{exp_special}           special |
{exp_var_name}          var_name variable_var_name? |
{exp_partence}          lpar exp rpar;

special =
{math_sinus}    sin lpar exp rpar |
{math_cosinus}  cos lpar exp rpar |
{math_random}   random lpar [exp1]:exp coma [exp2]:exp rpar
     |
{math_power}    power lpar [exp1]:exp coma [exp2]:exp rpar
     |
{math_sqrt}     sqrt lpar exp rpar |
{math_health}   gethealth lpar rpar |
{math_getview}  getview lpar rpar |
{math_canshoot} canshoot lpar rpar;

variable_var_name =
```

```
{exp_methos_call}       method_call |
{exp_id_list_call}      id_list_call;

types =
{var_real}      real |
{var_bool}      bool |
{var_string}    string;

event =
T.event event_name ltub commands rtub;

event_name =
{event_onhit}           onhit lpar var_real var_name
                        rpar |
{event_ondeath}         ondeath lpar var_string
                        var_name rpar |
{event_oncollosion}     oncollision lpar
                        [real1]:var_real
                        [var1]:var_name coma
                        [real2]:var_real
                        [var2]:var_name rpar |
{event_onsight}         onsight lpar [real1]:var_real
                        [var1]:var_name [coma1]:coma
                        var_bool [var2]:var_name
                        [coma2]:coma [real2]:var_real
                        [var3]:var_name rpar |
{event_ontargethit}     ontargethit lpar rpar |
{event_onstart}         onstart lpar rpar;

event_list =
event_list_name event_list_tail*;

event_list_tail =
coma event_list_name;

event_list_name =
{event_list_onhit}              onhit |
{event_list_ondeath}            ondeath |
{event_list_oncollision}        oncollision |
{event_list_onsight}            onsight |
{event_list_ontargethit}        ontargethit |
{event_list_all}                event_all;

method =
T.method method_input;

method_input =
{method_input_return}           return_type var_name lpar
                                parameters? rpar ltub
                                commands return exp
                                semicolon rtub |
{method_input_no_return}        var_name lpar parameters?
                                rpar ltub commands rtub;

return_type =
identifier;

parameters =
parameter parameters_tail*;

parameters_tail =
coma parameter;
```

```
parameter =
identifier var_name;

method_call =
lpar  method_inputs? rpar;

method_inputs =
exp input_tail*;

input_tail =
coma exp;

commands =
command*;

command =
{command_action}          action |
{command_if}              if_exp |
{command_while}           while_exp |
{command_select}          select_exp |
{command_assignment}      assignment |
{command_mathod_call}     var_name method_call
                          semicolon |
{command_type_decl}       type_decl ;

action =
action_name rpar semicolon;

action_name =
{action_ignore}                  ignore lpar event_list
                                 rpar |
{action_unignore}                unignore lpar
                                 event_list |
{action_rotate}                  rotate lpar exp |
{action_rotateleft}              rotateleft lpar |
{action_rotateright}             rotateright lpar |
{action_look}                    look lpar exp |
{action_lookleft}                lookleft lpar |
{action_lookright}               lookright lpar |
{action_strafe}                  strafe lpar exp |
{action_strafeleft}              strafeleft lpar |
{action_straferight}             straferight lpar |
{action_move}                    move lpar exp |
{action_moveforward}             moveforward lpar |
{action_movebackward}            movebackward lpar |
{action_chooseweapon}            chooseweapon lpar exp |
{action_shoot}                   shoot lpar exp? |
{action_say}                     say lpar exp |
{action_stop}                    stop lpar |
{action_stopmove}                stopmove lpar |
{action_stoprotate}              stoprotate lpar |
{action_stoplook}                stoplook lpar |
{action_stopshoot}               stopshoot lpar |
{action_wait}                    wait lpar |
{action_waitformove}             waitformove lpar |
{action_waitforrotate}           waitforrotate lpar |
{action_waitforlook}             waitforlook lpar |
{action_waitforshoot}            waitforshoot lpar |
{action_getviewingangle}         getviewingangle lpar |
{action_canshoot}                canshoot lpar |
{action_sleep}                   sleep lpar exp;
```

```

if_exp =
if exp ltub commands rtub else_exp ?;

else_exp =
else ltub commands rtub;

select_exp =
select exp ltub cases rtub;

cases =
case * P.default ?;

case =
case_exp ltub commands rtub;

default =
T.default ltub commands rtub;

case_exp =
case_exp_input case_exp_tail *;

case_exp_tail =
coma case_exp_input;

case_exp_input =
{case_set}     [bra1]:case_exp_bra [real1]:real colon
               [real2]:real [bra2]:case_exp_bra |
{case_real}    real;

case_exp_bra =
{case_exp_bra_left}     lbra |
{case_exp_bra_right}    rbra;

while_exp =
while exp ltub commands rtub;
```

Bibliography

[Alp] IBM AlphaWorks. Robocode. http://robocode.alphaworks.ibm.com/home/home.html. Seen 20.05.04.

[Bac86] Roland C. Backhouse. *Program Construction and Verification*. Prentice-Hall, 1986.

[CFR93] Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. *Program Verification, Fundamental Issues in Computer Science*. Kluwer Academic Publishers, 1993.

[fS96] International Organization for Standardization. Iso 14977. http://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf, 1996. Seen 15.04.04.

[Gag] Etienne M. Gagnon. Sablecc documentation. http://www.sablecc.org/documentation.php. Seen 18.05.04.

[Hüt04] Hans Hüttel. *Pilen ved træets rod*. Aalborg Universitet, 2004.

[ISO] International organization of standardization. Seen 28.04.04.

[java] Java community processsm maintenance review for j2setm 1.4.2 beta. http://java.sun.com/j2se/1.4.2/jcp/beta/. Seen 25.05.04.

[javb] Java technology. http://java.sun.com. Seen 14.05.04.

[Kir02] Andrew Kirmse. Optimization for c++ games. 2002.

[LY99a] Tim Lindholm and Frank Yellin. *The Java(tm) Virtual Machine Specification*, chapter 4 - The class file format. Sun Microsystems, Inc., http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html, 2. edition, 1999. Seen 07.05.04.

[LY99b] Tim Lindholm and Frank Yellin. *The Java(tm) Virtual Machine Specification*, chapter 6 - The Java Virtual Machine Instruction Set. Sun Microsystems, Inc., http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions.doc.html, 2. edition, 1999. Seen 07.05.04.

[Mb04] Kahled Mardam-bey. Frequently asked questions about mirc. http://www.mirc.com/faq7.html#section7, 2004. Seen 02.05.04.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications*. Wiley Professional Computing, 1999. Seen 28.04.04.

[Rea02] Steve Rabin and et al. *AI Programming Wisdom*. Charles River Media, INC, 2002.

[Seb04] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, international, sixth edition, 2004.

[Sip96] Michael Sipser. *Intoduction to the Theory of Computation*. Brooks Cole, 1996.

[Swi] Java switch statement tutorial. http://java.sun.com/docs/books/-tutorial/java/nutsandbolts/switch.html. Seen 14.05.04.

[Tea] C210 Game Developement Team. The lagna language and the gamna game. http://www.cs.auc.dk/~esben/dat2/lagna/index.html. Can be found on #C210 at irc.quakenet.org.

[Uni04] Inc. Unicode. Unicode home page. http://www.unicode.org, 2004. Seen 11.05.04.

[Vix] Paul Vixie. Crontab. http://man.linuxquestions.org/index.php-?query=crontab&section=0&type=2. Seen 20.05.04.

[WB99] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Pearson Higher Education, 1999.

[WB00] David A. Watt and Deryck F. Brown. *Programming Language Processors In Java*. Prentice Hall, 2000.